

Otto-Friedrich-Universität Bamberg

Lehrstuhl für Praktische Informatik



Integration heterogener prozessbasierter Systeme in Service–orientierten Architekturen – Eine Petri–Netz–basierte Integration von Windows Workflow und BPEL

Stefan Kolb

Juli 2012

Inhaltsverzeichnis

1 Einleitung	1
2 Prozessbasierte Systeme	3
2.1 Workflowsprachen	4
2.1.1 Interaktion über Web Services	4
2.1.2 Business Process Execution Language	4
2.1.3 Windows Workflow Xaml Vocabulary	6
2.2 Workflow Engines	9
2.2.1 Windows Workflow Foundation	10
2.2.2 Apache Orchestration Director Engine	11
3 Serviceinteraktion mit Petri–Netzen	14
3.1 Petri–Netze	14
3.2 Komposition	16
3.3 Integration und Adaption	19
3.3.1 Adapterspezifikation	20
3.3.2 Adaptersynthese	22
4 Patterns	25
4.1 Sequentielle Workflows	27
4.1.1 Primitive Aktivitäten	27
4.1.2 Strukturierte Aktivitäten	29
4.2 Flowchart Workflows	34
4.2.1 FlowDecision	35
4.2.2 FlowSwitch<T>	36
4.3 StateMachine Workflows	37
4.3.1 State	38
4.3.2 Transition	39
4.4 Einschränkungen	40

5 Compiler Prototyp WF2oWFN	41
5.1 Architektur	41
5.1.1 API	42
5.1.2 Compiler	44
5.2 Entwicklung eines Moduls	47
6 Fallbeispiele	51
6.1 Restaurant–Prozess	52
6.1.1 Tourist	52
6.1.2 Koch	55
6.1.3 Adaptersynthese	56
6.1.4 Integration	60
6.2 UBL Ordering–Prozess	62
6.2.1 Seller Party	63
6.2.2 Buyer Party	68
6.2.3 Adaptersynthese und Integration	70
7 Future Work	75
8 Fazit	76
A Xaml(x)–Spezifikation	77
A.1 Primitive Aktivitäten	80
A.1.1 WriteLine	80
A.1.2 Delay	80
A.1.3 Assign	80
A.1.4 Receive	80
A.1.5 ReceiveReply	81
A.1.6 Send	81
A.1.7 SendReply	81

INHALTSVERZEICHNIS

A.2 Strukturierte Aktivitäten	82
A.2.1 Sequence	82
A.2.2 Parallel	82
A.2.3 Pick	82
A.2.4 While	82
A.2.5 DoWhile	83
A.2.6 If	83
A.2.7 Switch<T>	83
A.3 Flowchart	84
A.3.1 FlowDecision	84
A.3.2 FlowSwitch<T>	85
A.4 StateMachine	85

Apache ODE	Apache Orchestration Director Engine
API	Application Programming Interface
BPEL	Business Process Execution Language
BPMN	Business Process Model and Notation
CDATA	Character Data
CLR	Common Language Runtime
CIL	Common Intermediate Language
DOM	Document Object Model
DoS	Denial of Service
HTTP	Hypertext Transfer Protocol
IIS	Internet Information Services
IT	Informationstechnologien
OSP	Microsoft Open Specification Promise
oWFN	Open Workflow Nets
OASIS	Organization for the Advancement of Structured Information Standards
PNG	Portable Network Graphics
PAIS	Process–Aware Information Systems
SEA	Specification of the Elementary Activities
SOAP	ursprünglich für Simple Object Access Protocol
SOA	Service–orientierte Architektur
UBL	Universal Business Language
XPath	XML Path Language
Xaml	Extensible Application Markup Language
XSD	XML Schema Definition
XSLT	Extensible Stylesheet Language Transformation
WSDL	Web Services Description Language
WCF	Windows Communication Foundation
WF	Windows Workflow (Foundation)
WPF	Windows Presentation Foundation
W3C	World Wide Web Consortium
WFXV	Windows Workflow Xaml Vocabulary

Abbildungsverzeichnis

1	Übersicht der Windows Workflow Foundation Architektur [Mic07]	10
2	Gast–Service GS_1 und Restaurant–Service RS_1 [VMSW09, S.3]	16
3	Gast–Service GS_1 und Restaurant–Service RS_2 [VMSW09, S.3]	17
4	Gast- und Restaurant-Services $GS_2 \otimes RS_3$ und $GS_3 \otimes RS_4$ [VMSW09, S.4]	18
5	Integration mit einem Adapter-Service [VMSW09, S.39]	19
6	Konzeptuelle Adapterstruktur [VMSW09, S.39]	22
7	Controller und Engine des Beispieladapters	23
8	Interface–Pattern der Aktivitäten	26
9	Patterns der primitive Aktivitäten	27
10	Patterns der Nachrichtenaktivitäten	28
11	Pattern der Sequence–Aktivität	29
12	Pattern der Parallel–Aktivität	30
13	Pattern der If–Aktivität	30
14	Pattern der Switch–Aktivität	31
15	Pattern der Pick–Aktivität	32
16	Patterns der Schleifen–Aktivitäten	33
17	Beispiel–Pattern eines Flowcharts	34
18	FlowStep –Pattern	35
19	FlowDecision –Pattern	35
20	FlowSwitch –Pattern	36
21	Beispiel–Pattern einer StateMachine	37
22	State –Pattern	38
23	Transition –Patterns	39
24	Grobübersicht der Funktionsweise des Compilers	41
25	Das API von WF2oWFN	43
26	Der Compiler WF2oWFN	45
27	Abläufe, Artefakte und Tools der Integrationsprozesse	51

ABBILDUNGSVERZEICHNIS

28	WF–Prozess Tourist	52
29	BPEL–Prozess Koch	55
30	Liberalster Adapter Restaurant	56
31	oWFN– und BPEL–Adapter Restaurant	60
32	UBL Ordering–Prozess [OAS11, S.28]	62
33	WF Ordering–Seller	64
34	oWFN Ordering–Seller	67
35	BPEL Ordering–Buyer	68
36	oWFN–Netze Buyer	70
37	oWFN Ordering–Adapter	72
38	WF Ordering–Mediator	73

Tabellenverzeichnis

1	SEA des Restaurant–Adapters [VMSW09, S.32]	22
2	SEA Ordering–Adapter	71

1 Einleitung

Die Entwicklung der *Informationstechnologien (IT)* hat die Geschäftsprozesse, innerhalb und über Unternehmensgrenzen hinweg, nachdrücklich verändert. Traditionelle IT diente meist dem Zweck eine bestimmte Aufgabe zu erfüllen, die heutigen Geschäftsprozesse und ihre IT sind jedoch häufig stark verflochten. Die Prozesse hängen von den Informationssystemen ab und diese sind wiederum getrieben von den Prozessen, die sie unterstützen. In den letzten zehn Jahren sind die Informationssysteme „process-aware“ geworden, d.h. Prozesse gelten als Startpunkte einer Implementierung (vgl. [DvT05]). Solche *Process-Aware Information Systems (PAIS)* (engl. für Prozessbasierte Systeme), werden in der aktuellen wissenschaftlichen Diskussion rege besprochen. Gleichzeitig gibt es eine steigende Akzeptanz von *Service-orientierten Architekturen (SOA)* als ein Paradigma für die Integration von Softwareanwendungen (vgl. [ACKM04]). Web Services–Technologien, wie SOAP und die *Web Services Description Language (WSDL)*, erleichtern die Realisierung solcher lose gekoppelter Architekturen. SOA und die verbundenen Technologien haben die klassische Unterscheidung zwischen intra- und interorganisationalen Prozessen verschwimmen lassen (vgl. [VMSW09]). Um die Potentiale der Geschäftsprozesse eines Unternehmens voll auszuschöpfen, gilt es die verschiedenen beteiligten heterogenen Prozesssysteme zu integrieren. Dies können einerseits eigene Anwendungen der Firmen sein, die speziell auf die unternehmensinternen Prozesse abgestimmt sind, andererseits eine Reihe von generischen Produkten zur Ausführung von Prozesssprachen. Im Bereich der Web Services–basierten Sprachen stellt die *Business Process Execution Language (BPEL)* aktuell den de facto Standard für Prozessmodellierung dar. Neben diesem offenen Standard existieren einige weitere Sprachen, von denen gerade im Windows–Segment *Windows Workflow (WF)* zu nennenswerter Popularität gelangt ist. Durch die Nutzung von Web Services existiert ein gemeinsamer Standard für die Kommunikation zwischen den Systemen. Während die Technologie zur Entwicklung einfacher Services und zur Verknüpfung derselben einen gewissen Reifegrad erreicht hat, bleiben offene Herausforderungen, wenn es dazu kommt Services zu arrangieren, die komplexe Interaktionen mit mehreren anderen Partnerservices durchführen (vgl. [OVVDA⁺07, S.162]). Gerade bei prozessgetriebenen Systemen stellt sich neben den Kommunikationsaspekten vor allem die Frage der Kontrollflusskonformität der zu integrierenden Systeme, d.h. es ist zu klären, ob die Systeme korrekt miteinander interagieren und die Abläufe der Geschäftsprozesse zueinander passen. Wenn nicht, gilt es diese Probleme zu lösen, um einen korrekten störungsfreien Prozessablauf zu ermöglichen. Da Geschäftsprozesse meist sehr komplexe Prozesse abbilden, stellt sich die Anforderung dies möglichst automatisiert zu tun. Hierfür bedarf es einer fundierten Grundlage, um die auftretenden Probleme und Inkompabilitäten zu analysieren und zu überwinden. Da die meisten Prozessmodellierungssprachen, so auch BPEL und WF, informell definiert sind und nicht auf methodischen Grundlagen basieren, müssen diese zunächst auf eine analysierbare Modellierungsebene abgebildet werden. Petri–Netze haben sich zum Zwecke der Modellierung und Analyse von Geschäftsprozessen bewährt (vgl. [Van98, Vv04]). Mit *Open Workflow Nets (oWFN)* [MRS05] steht zudem eine Verfeinerung der bekannten klassischen Petri–Netze im Kontext der Servicemodellierung zur Verfügung, die bereits in einer ausführlichen Betrachtung von BPEL ihr Potential gezeigt hat (vgl. [Loh07, OVVDA⁺07]). Van der Aalst [Van98] betrachtet zudem diverse Szenarien der Serviceinteraktion mit Petri–Netzen und Konzepte zur Lösung typischer Integrationsprobleme.

Zweck der Arbeit ist es, Probleme und Lösungswege der Integration von heterogenen Prozesssystemen aufzuzeigen, im Speziellen zwischen BPEL und WF. In [Loh07] wurde bereits eine Petri–Netz–Semantik für BPEL vorgestellt, auf deren Grundlage eine äquivalente Semantik für WF erarbeitet werden soll. Aufgrund der Zeitbeschränkung wird jedoch in der vorliegenden Arbeit keine vollständige Semantik für WF konzipiert, sondern nur ein Ausschnitt der Funktionalität. Zudem soll nur der Gutfall, d.h. der fehlerfreie Kontrollfluss betrachtet werden. Aufgrund des verwendeten Petri–Netz–Formalismus werden außerdem alle Daten– und Zeitaspekte der Prozesse abstrahiert. Anschließend soll diese Semantik in einen Prototypen implementiert werden, der WF–Prozesse automatisiert in oWFN transformiert. Zusammen mit einer bereits bestehenden Implementierung zur Transformation von BPEL–Prozessen und verschiedenen Tools zur Analyse von Verhaltensaspekten von Petri–Netzen sowie zur Adaptersynthese, wird dann anhand zweier Fallbeispiele die automatisierte Integration von Prozessen der beiden Prozesssysteme evaluiert. Dabei soll vor allem die Kontrollflusskonformität der Integration sichergestellt werden, jedoch auch mögliche Kommunikationprobleme mit dargelegt werden. Hier wird zum einen auf ein konzeptuelles Beispiel aus [VMSW09] zurückgegriffen, um erwartete Ergebnisse zu bestätigen und mögliche Probleme in der Interoperabilität der Systeme zu betrachten, zum anderen soll ein größeres Realbeispiel aus [OAS11] zum Einsatz kommen, um eine realitätsnahe Evaluation der Konzepte zu ermöglichen.

Zunächst werden die Grundlagen der verschiedenen Aspekte der Arbeit dargestellt, um ein besseres Verständnis der späteren Betrachtungen zu ermöglichen. Kapitel 2 beschäftigt sich deswegen eingangs mit prozessbasierten Systemen. Neben einer Definition fundamentaler Anforderungen und Teilaspekten dieser Systeme, werden die beiden thematisierten Workflowsprachen BPEL und WF genauer betrachtet. Zusätzlich werden Informationen und Grundkonzepte der zugehörigen Laufzeitumgebungen erläutert. Kapitel 3 stellt den gewählten Petri–Netz–Formalismus und Konzepte der Serviceinteraktion mit Petri–Netzen vor. Es zeigt, wie typische Szenarien der Interaktion und Integration mithilfe von Petri–Netzen formal gelöst werden können. Kapitel 4 stellt die Funktionsweise der Konstrukte von WF und die erarbeiteten Petri–Netz–Repräsentationen vor. Anschließend werden die Konzepte und Möglichkeiten des entwickelten Prototypen in Kapitel 5 erläutert. Kapitel 6 zeigt die praktische Anwendung der Konzepte und Tools anhand zweier Fallbeispiele. Schlussendlich werden noch Verbesserungsmöglichkeiten aufgezeigt und ein Fazit schließt die Arbeit ab.

2 Prozessbasierte Systeme

Ein PAIS ist ein beliebiges Softwaresystem, das operative Prozesse (*Workflows*) auf der Basis von Prozessmodellen verwaltet und ausführt, an denen Personen, Anwendungen oder andere Informationsquellen beteiligt sein können (vgl. [DvT05, S.7]). Da die Kosten und die Skalierbarkeit einer maßgeschneiderten Lösung für die meisten Organisationen zu teuer und ineffizient sind, existieren generische PAIS. Um solche Systeme zu nutzen, muss ein Unternehmen seine Prozesse in einem vom PAIS unterstützten Prozessmodell definieren. Hierfür existieren Workflowsprachen, die das Prozessmodell in einem maschinenlesbaren Format abbilden. Die so definierten Prozesse werden anschließend vom PAIS ausgeführt. Dadurch, dass prozessbasierte Systeme meist von Prozessmodellen statt von Code getrieben sind, ermöglichen sie es Geschäftsprozesse zu ändern, ohne größere Teile des Systems neu zu implementieren. Das Prozessmodell beinhaltet die Geschäftslogik, d.h. die Ausführungsreihenfolge der einzelnen Arbeitsschritte. Es trennen dabei explizit die Logik von der Implementierung der Prozesse (vgl. [DvT05, S.7–10]). Auch wenn viele verschiedene Prozessmodelle existieren, so finden sich doch einige essentielle Elemente und Konzepte automatisierbarer Prozesse in allen wieder. Die grundlegenden Konstrukte eines Prozesses sind Aufgaben oder Aktivitäten. Das Prozessmodell bildet die Kontrollfluss- und Datenflussabhängigkeiten zwischen verschiedenen Aktivitäten ab (vgl. [RtEv05, S.354 ff.]). Die meisten Workflowsprachen stellen diesen Ablauf entweder über einen gerichteten Graphen oder in einer blockstrukturierten, sequentiellen Form dar (vgl. [KMWL09]). Eine konkrete Instanz eines Prozessmodells heißt Prozessinstanz. Mehrere Instanzen des selben Prozesses können parallel und unabhängig voneinander ausgeführt werden. Während der Ausführung einer Prozessinstanz werden die definierten Aktivitäten instanziert und dem Kontrollfluss nach ausgeführt. Eine Aktivität stellt eine einzelne Arbeitseinheit dar. Es existieren dabei drei unterschiedliche Aktivitätstypen: Zum einen primitive Aktivitäten, die abgeschlossene Operationen durchführen, zum anderen strukturierte Aktivitäten, die weitere Aktivitäten beinhalten und deren Ausführung in einer Art Subprozess realisieren. Letztendlich gibt es Mehrinstanzaktivitäten, die mehrere unabhängige parallele Ausführungen einer identischen Aktivitätsdefinition innerhalb einer Prozessinstanz ermöglichen (vgl. [RtEv05, S.355 f.]). In Service-orientierten Architekturen sind die Services idealerweise dafür ausgelegt, komplexe Prozesse zu implementieren (vgl. [Pel03]). Konkret werden von einem Prozess eine Reihe verschiedener Services aufgerufen und Nachrichten ausgetauscht. Dabei können wie oben genannt mehrere Prozessinstanzen innerhalb eines PAIS existieren. Das PAIS benötigt deshalb eine Möglichkeit, um die eingehenden Nachrichten an die korrekte Empfängerinstanz weiterzuleiten. In objektorientierten, eng gekoppelten Systemen wird dies durch die Weitergabe von Objektreferenzen verwirklicht. In einer losen gekoppelten SOA würde dies jedoch zu großen Abhängigkeiten zwischen den Systemen führen, welche durch Veränderungen in den Teilsystemen leicht gebrochen werden könnten. Deshalb findet die Korrelation in Service-orientierten Architekturen auf Nachrichtenebene statt. Diese Technik wird als Nachrichtenkorrelation bezeichnet (vgl. [OAS07, S.74 ff.]). Die Implementierung dieser Korrelation ist dabei sprachspezifisch. Im Zuge von Webservices sind hier kontextbasierte und inhaltsbasierte Korrelation üblich. Kontextbasierte Korrelation verwendet Standards wie WS-Adressing [W3C07b], um Kontextinformationen auszutauschen. Hierbei müssen beide Partner jedoch auch den gleichen Standard anwenden um kompatibel zu sein. Die unabhängige Technik ist deshalb die inhaltsbasierte Korrelation. Hierbei enthält eine Nachricht Datenelemente, die eine eindeutige Zuordnung zu einer Prozessinstanz durch die Laufzeitumgebung erlauben.

2.1 Workflowsprachen

Mit einer Workflowsprache wird ein Prozessmodell in einem maschinenlesbaren Format abgebildet. Die meisten Sprachen verwenden hierzu eine XML-basierte Repräsentation. Der im Kontext von Service-orientierten Architekturen häufig benutzte Begriff der Orchestrierungssprache stellt dabei eine Unterkategorie der Workflowsprachen dar. Der Begriff Orchestrierung beschreibt in diesem Zusammenhang einen ausführbaren Geschäftsprozess, der ausschließlich über Services interagiert (vgl. [Pel03, S.46]). Im Gegensatz dazu sind die Prozesse eines PAIS nicht zwangsläufig servicebasiert. Da alle Interaktionen der Systeme in dieser Arbeit jedoch über Service abgewickelt werden sollen, im Speziellen über Webservices, werden zunächst kurz deren Grundlagen angesprochen. In der Folge sollen die beiden in der Arbeit thematisierten Sprachen, BPEL und WF, sowie deren grundlegende Konzepte vorgestellt werden.

2.1.1 Interaktion über Web Services

Web Services stellen eine häufig genutzte Technologie für die Implementierung von Service-orientierten Architekturen dar. Die verschiedenen zugehörigen Standards werden vom *World Wide Web Consortium (W3C)* verwaltet. Ein Webservice ist ein Softwaresystem, das darauf ausgelegt ist die interoperable Kommunikation zwischen Systemen über das Internet mit Hilfe der Web Standards zu unterstützen. Web Services nutzen ein lose gekoppeltes Integrationsmodell, um die flexible Integration von heterogenen Systemen in beliebigen Anwendungsdomänen zu ermöglichen. Zu diesem Zweck besitzt ein Webservice ein Interface, das durch die WSDL beschrieben wird (vgl. [W3C04]). Die WSDL des Webservices definiert neben den möglichen Operationen innerhalb der **portTypes** auch die Typen (**types**) der zu übertragenden Nachrichten. Diese können sowohl in der WSDL definiert werden, als auch in sog. *XML Schema Definition (XSD)*-Dateien ausgelagert werden. Das **binding** legt das Nachrichtenformat und das Transportprotokoll der verschiedenen Operationen eines **portTypes** fest. Durch den **port** wird ein **binding** einer spezifischen Endpoint-Adresse zugewiesen, unter der der Webservice erreichbar ist. Das **service**-Element gruppiert eine Menge von zusammengehörigen **ports** (vgl. [W3C01]). Andere Systeme interagieren über die Definition des Interfaces mit dem Webservice durch Nutzung von SOAP-Nachrichten, welche typischerweise über das *Hypertext Transfer Protocol (HTTP)* übertragen werden. SOAP ist eine XML-basierte Darstellung von Nachrichten. Eine SOAP-Nachricht besteht dabei aus einem sog. *Envelope*, der einen optionalen Nachrichtenkopf (*Header*) und den Nachrichteninhalt (*Body*) enthält. Im Nachrichtenkopf können Metadaten, wie Verschlüsselung oder Korrelation, übertragen werden. Der Aufbau des Nachrichteninhalts für eine Operation muss der in der WSDL definierten Nachrichtenstruktur entsprechen (vgl. [W3C07a]).

2.1.2 Business Process Execution Language

BPEL ist eine XML-basierte Orchestrierungssprache zur Beschreibung von Geschäftsprozessen. Aktuell wird der BPEL-Standard von der *Organization for the Advancement of Structured Information Standards (OASIS)* verwaltet und ist in der Version 2.0 [OAS07] frei verfügbar. BPEL baut auf den Web Services Standards WSDL 1.1, XML Schema 1.0, XPath 1.0 und XSLT 1.0 auf. WSDL-Nachrichtentypen und XSDs stellen dabei das

Datenmodell der BPEL-Prozesse dar. XPath und XSLT werden für die Datenmanipulation benutzt. BPEL ist die Orchestrierungssprache, die aktuell die größte Verbreitung erreicht hat und als de facto Standard für Orchestrierungen gilt. Alle externen Ressourcen und Partner werden als WSDL-Services repräsentiert. BPEL definiert eine Menge an Kontrollflusskonstrukten wie Bedingungen, Schleifen sowie Aktivitäten, um Nachrichten an Webservices zu senden als auch zu empfangen. BPEL stellt vornehmlich sequentielle Aktivitäten zur Verfügung, wobei auch eine graphbasierte Modellierung über das `flow`-Konstrukt möglich ist. Eine genau Auflistung der verfügbaren Aktivitäten würde den Rahmen der Arbeit sprengen. Die Konstrukte ähneln des Weiteren zum großen Teil denen der in WF verfügbaren, die später noch detailliert vorgestellt werden. Für eine Auflistung und genaue Betrachtung aller Aktivitäten sei deshalb auf die BPEL-Spezifikation [OAS07] verwiesen. BPEL-Prozesse können sowohl als abstrakte Prozesse als auch als ausführbare Prozesse existieren. Abstrakte Prozesse dienen dabei der Beschreibung eines allgemeinen Prozessmodells mehrerer Anwendungsfälle und sind nicht ausführbar. Ausführbare Prozesse hingegen sind vollständig spezifizierte Prozesse für einen konkreten Anwendungsfall (vgl. [OAS07, S.7 f.]). Ein BPEL-Prozess besteht neben dem durch die Aktivitäten dargestellten Kontrollfluss aus weiteren notwendigen Elementen. Die grundlegende Struktur eines BPEL-Prozesses wird in Listing 1 aufgezeigt.

Listing 1: Allgemeine Struktur eines BPEL-Prozesses (nach [Len11, S.8])

```

1 <process name="SampleProcess">
2   <!-- Imports -->
3   <import location="MyRole.wsdl"
4     importType="http://schemas.xmlsoap.org/wsdl/" />
5   <!-- PartnerLinks -->
6   <partnerLinks>
7     <partnerLink name="MyRolePartnerLink"
8       partnerLinkType="MyRolePartnerLinkType" myRole="myRole" />
9   </partnerLinks>
10  <!-- Variables -->
11  <variables>
12    <variable name="InputParameter" messageType="InputMessage" />
13  </variables>
14  <!-- CorrelationSets -->
15  <correlationSets>
16    <correlationSet name="CorrelationSet"
17      properties="PropertyFromWSDL" />
18  </correlationSets>
19  <!-- Orchestration -->
20  <sequence name="ProcessFlow">
21    <receive name="StartProcess" createInstance="yes"
22      variable="InputParameter" partnerLink="MyRolePartnerLink"
23      operation="OperationFromWSDL" />
24  </sequence>
25 </process>
```

Alle Elemente des Prozesses sind dabei innerhalb eines `process`-Elements beherbergt. Es folgen Import-Anweisungen, die zumindest die eigene WSDL des Services importieren müssen. Zusätzlich werden alle weiteren Definitionen der Partnerprozesse importiert. Um in BPEL andere Webservices aufzurufen oder eigene Services anzubieten, müssen entsprechende `partnerLinks` im Prozess definiert werden. Ein `partnerLink` hat einen Namen, über den er für alle Interaktionen, die über ihn abgewickelt werden, referenziert

wird. Die Rolle des Prozesses selbst, in dem er deklariert wurde, wird durch das Attribut *myRole* beschrieben sowie die Rolle eines Partners über das Attribut *partnerRole*. Eine oder beide Rollen müssen gesetzt sein. Jeder einzelne *partnerLink* wird durch einen *partnerLinkType* beschrieben, der sich in der WSDL des jeweilig zugehörigen Services befinden muss und einen *portType* referenziert (vgl. [OAS07, S.36–39]). Ein Prozess enthält immer genau eine Aktivität, die den gesamten Kontrollfluss des Prozesses beinhaltet.

Geschäftsprozesse spezifizieren zustandsbehaftete Interaktionen, die Nachrichten zwischen den Partnern mit einbeziehen. Der Zustand eines Prozesses wird durch die ausgetauschten Nachrichten als auch durch die Daten die in dem Prozess genutzt werden bestimmt. Die Verwaltung des Zustandes wird über die Nutzung von Variablen realisiert. Die in den Variablen enthaltenen Daten können dann durch Aktivitäten extrahiert, kombiniert oder manipuliert werden, um das Verhalten des Prozesses zu kontrollieren (vgl. [OAS07, S.45 ff.]).

Um eingehende Nachrichten einer korrekten Prozessinstanz durch inhaltsbasierte Korrelation zuzuordnen, existieren in BPEL sog. **correlationSets**. Innerhalb eines **correlationSets** wird auf eine Eigenschaft verwiesen, die für die Korrelation verwendet werden soll. Diese Eigenschaft, die sog. **property**, wird in der zugehörigen WSDL des Services definiert und muss von einem primitiven Typ sein. Für jede **property** müssen für alle Nachrichtentypen, welche in Aktivitäten zur Verwendung kommen und an der Korrelation teilnehmen, jeweils sog. **propertyAliases** definiert werden. Hierdurch findet mit Hilfe einer XPath–Query eine Abbildung auf die konkreten Variablen der Nachrichten statt. An der ersten **receive**–Aktivität wird das **correlationSet** referenziert und mit dem Attribut *initiate=true* angewiesen, eine neue Instanz zu initiieren. An allen weiteren korrelierten Aktivitäten wird das Attribut auf **false** gesetzt, damit versucht wird eine Korrelation mit einer bereits initialisierten Instanz herzustellen (vgl. [OAS07, S.40–44, 74–83]).

2.1.3 Windows Workflow Xaml Vocabulary

Im Gegensatz zu BPEL ist WF kein offener Standard, sondern wird von Microsoft als Teil des .NET Frameworks entwickelt. Das .NET Framework ist aktuell in der Version 4 Platform Update 1¹ verfügbar. Alle in der Arbeit dargelegten Aspekte basieren auf dieser Version. WF ist zudem fest in die Entwicklungsumgebung Visual Studio inklusive eines visuellen Workflow–Designers und einer Laufzeitumgebung innerhalb der Windows Workflow Foundation eingebettet. Da große Teile der Arbeit jedoch vor allem auf der Sprachebene basieren, wird diese trotzdem konzeptuell abgetrennt. Zu den Laufzeitaspekten wird im Kapitel 2.2 genaueres ausgesagt.

Die Markupsprache von WF ist de facto keine Orchestrierungssprache wie BPEL. Zum einen wird ein Prozess in WF nicht zwangsläufig als Service veröffentlicht, sondern kann in beliebigen Anwendungen innerhalb des .NET Frameworks und mit einer Reihe von Kommunikationstechnologien integriert werden, noch handelt es sich beim Framework um eine servicebasierte Architektur. Dennoch bietet das Framework alle Möglichkeiten, um in eine SOA eingebunden zu werden. Im Zuge der Arbeit werden die Interaktionen aller beteiligten Partner ausschließlich über Webservices als Teile einer SOA betrachtet, was für WF

¹Siehe <http://msdn.microsoft.com/en-us/library/hh290669>

eine Teilmenge der Fähigkeiten darstellt. Basis der Workflowsprache in WF ist die *Extensible Application Markup Language (Xaml)*. Xaml ist eine deklarative XML-basierte Markupsprache zur Repräsentation von strukturierten Informationen. Deren Spezifikation definiert zwei abstrakte Informationsmodelle: Das *Xaml Schema Information Set* und das *Xaml Information Set*. Das Xaml Information Set definiert die Struktur der Informationen, die eine Xaml-Instanz repräsentieren kann. Das Xaml Schema Information Set erlaubt es, spezifische Xaml-Vokabulare zu verfassen. Anwendungen können dadurch ihr eigenes Vokabular für ihren spezifischen Anwendungsbereich auf Grundlage der Xaml-Spezifikation definieren. Ein Schema definiert dabei die Objekttypen, die in der Xaml-Instanz genutzt werden können, d.h. die Elemente und Inhalte. Die Xaml-Spezifikation spezifiziert jedoch kein spezielles Darstellungsformat für diese Schemata. Diese werden von Microsoft aktuell formlos als Word-Dokumente dargestellt. Das Xaml Schema Information Set legt jedoch fest, welche Informationen für ein komplettes Schema vonnöten sind (vgl. [Mic12r, S.8]). WF nutzt eben diese Basis, um ein eigenes Vokabular zu definieren, das eine deklarative Darstellung der WF-Prozesse ermöglicht. Zumal es sich bei WF um ein proprietäres System handelt, sind keine Spezifikationen zum Aufbau des Xaml-Vokabulars verfügbar. Da eine solche Definition jedoch für die Ziele der Arbeit nötig ist, wurde deshalb im Verlauf der Arbeit eine eigene Spezifikation für die implementierten Aktivitäten erarbeitet (siehe Anhang A). Im März 2012 wurde die zitierte Xaml-Basis spezifikation und die konkreten Vokabulare der *Windows Presentation Foundation (WPF)* und von Silverlight unter der *Microsoft Open Specification Promise (OSP)* [Mic06], d.h. für jedermann frei nutzbar, veröffentlicht. Das Vokabular für WF indessen findet sich nicht unter den publizierten Spezifikationen (vgl. [Mic12e]). Eine Freigabe der Spezifikation würde ganz neue Möglichkeiten eröffnen (siehe auch Kapitel 7). Folgt man der internen Bezeichnung der bereits veröffentlichten Vokabulare, so muss man die Workflowsprache von WF als *Windows Workflow Xaml Vocabulary (WF-XV)* bezeichnen. Da in der Arbeit nur dieses Vokabular thematisiert wird, wird es im Folgenden lediglich als *Xaml* bezeichnet. Abbildung 2 zeigt die grundlegende Struktur eines in Xaml deklarierten WF-Prozesses².

Listing 2: Allgemeine Struktur eines WF-Prozesses [Buk10, S.14]

```

1 <Activity x:Class="HelloWorkflow.Workflow1"
2   mva:VisualBasic.Settings="Assembly references and imported
      namespaces for internal implementation"
3   xmlns="http://schemas.microsoft.com/netfx/2009/xaml/activities"
4   xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility
      /2006"
5   xmlns:mv="clr-namespace:Microsoft.VisualBasic;assembly=System"
6   xmlns:mva="clr-namespace:Microsoft.VisualBasic.Activities;
      assembly=System.Activities"
7   xmlns:s="clr-namespace:System;assembly=mscorlib"
8   xmlns:s1="clr-namespace:System;assembly=System"
9   xmlns:s2="clr-namespace:System;assembly=System.Xml"
10  xmlns:s3="clr-namespace:System;assembly=System.Core"
11  xmlns:sad="clr-namespace:System.Activities.Debugger;assembly=
      System.Activities"
12  xmlns:scg="clr-namespace:System.Collections.Generic;assembly=
      System"
13  xmlns:scg1="clr-namespace:System.Collections.Generic;assembly=
      System.ServiceModel"

```

²Zusatzinformationen, die für die Ausführung des Prozesses nicht nötig sind, wurden entfernt.

```

14  xmlns:scg2="clr-namespace:System.Collections.Generic;assembly=
    System.Core"
15  xmlns:scg3="clr-namespace:System.Collections.Generic;assembly=
    mscorelib"
16  xmlns:sd="clr-namespace:System.Data;assembly=System.Data"
17  xmlns:sl="clr-namespace:System.Linq;assembly=System.Core"
18  xmlns:st="clr-namespace:System.Text;assembly=mscorlib"
19  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
20  <Sequence>
21      <WriteLine Text="Hello Workflow" />
22  </Sequence>
23 </Activity>
```

Vernachlässigt man zunächst die Namespaces, kann man den Kontrollfluss, bestehend aus einer `Sequence` und einer enthaltenen `WriteLine`-Aktivität, erkennen. Während für die `Sequence`-Aktivität einfach das Äquivalent in BPEL identifiziert werden kann, zeigt die `WriteLine`-Aktivität, dass WF auch einige Aktivitäten zur Verfügung stellt, die über serviceorientierte Interaktionen hinaus gehen. Im konkreten Fall schreibt die Aktivität den Text „Hello Workflow“ an die Konsolenausgabe. Im Gegensatz zu den meisten anderen Markupsprachen, die oft eine interpretierte Sprache darstellen, fordert Xaml ein direktes Mapping der Elemente auf Typen innerhalb der zugehörigen Laufzeit, um eine kompilierte Version des Prozesses zu erstellen. Dadurch werden Elemente auf Klassen und deren Attribute direkt auf entsprechende Eigenschaften abgebildet. Der Prozess wird dennoch vollständig in Xaml abgebildet, d.h. es werden keine zusätzlichen Codedateien benötigt. Gemäß der Xaml-Basissspezifikation könnten Xaml-Typen dabei auch einem beliebigen anderen Typsystem als dem des .NET Frameworks zugeordnet werden. Es wäre dann aber erforderlich, einen eigenen Xaml-Parser zu erstellen (vgl. [Mic12e]). Die vielen Namespaces in der Xaml werden dementsprechend in Importanweisungen umgesetzt, die von Teilen der Xaml-Elemente auf Objektebene genutzt werden. Die Laufzeit kompiliert den deklarativen Prozess direkt in eine Klasse. Der Namespace und Name der Klasse wird durch das Attribut `x:Class` definiert (vgl. [Buk10, S.14 f.]).

Alle Elemente des Prozesses befinden sich innerhalb des `Activity`-Elements. WF unterscheidet hier zwischen normalen Workflows (`Activity`) und speziellen `WorkflowServices`, die darauf ausgelegt sind, den Prozess ohne große Zusatzkonfiguration als Webservice zu veröffentlichen (z.B. im Windows App Server, IIS). Ein `WorkflowService` formt dabei immer eine Orchestration. Im Fallbeispiel werden deshalb alle WF-Prozesse als `WorkflowServices` implementiert werden. `WorkflowServices` müssen zumindest eine eingehende Nachrichtenaktivität haben, die eine Instanz des Workflows erstellen kann. In Xaml wechselt dabei das Wurzelement von `Activity` zu `WorkflowService`, ansonsten existieren keine großen Unterschiede bei der Modellierung der beiden Typen. `WorkflowServices` verwenden für die Kommunikation über Webservices die *Windows Communication Foundation (WCF)* des .NET Frameworks. WCF basiert dabei genauso wie BPEL auf WSDL 1.1. Die Struktur der eigenen WSDL wird automatisch aus den Nachrichtenaktivitäten und deren Datentypen des Prozesses abgeleitet. Die Nutzung von WCF ist für den Entwickler dabei transparent. Alle Konfigurationen können über die Xaml bzw. eine Konfigurationsdatei durchgeführt werden. In der Datei können alle Konfigurationen bezüglich der Webanwendung wie Bindings, Endpoints, usw. deklarativ definiert werden.

WF ermöglicht mit entsprechenden Aktivitäten die Modellierung von Prozessen in drei unterschiedlichen Modellierungsstilen: Zum einen die konventionelle sequentielle Model-

lierung. Darüber hinaus wird eine graphbasierte Modellierung über sog. Flowcharts ermöglicht. Schlussendlich ist es in WF auch möglich, Prozesse durch endlichen Automaten (**StateMachine**) zu definieren. Da alle Aktivitäten im Zuge der Petri–Netz–Pattern (siehe Kapitel 4) vorgestellt werden, wird hier nicht weiter auf sie eingegangen. In WF besteht des Weiteren die Möglichkeit, neue Aktivitäten hinzuzufügen. Solche **CustomActivities** können jederzeit durch Code oder durch deklarative Orchestrierung vorhandener Aktivitäten geschaffen werden (vgl. [Buk10, S.46 f.]).

Da für die Xaml keine öffentliche Spezifikation verfügbar ist, ist WF eng mit .NET und dem Visual Studio Designer verbunden. Alle Workflows werden dort interaktiv erstellt und auch alle Konfigurationen getätigt. Aus diesem Grund sind die Betrachtungen in diesem Fall auch weniger codebasiert wie bei BPEL. Im Gegensatz zu BPEL wird die Struktur der Nachrichten nicht komplett deklarativ definiert. WF unterstützt zur Definition von Nachrichtentypen standardmäßig zwei Möglichkeiten: **DataContract** und **MessageContract**. **DataContract** beschreibt die Inhaltsstruktur der Nachrichten, während **MessageContract** es ermöglicht, sowohl die Struktur des Nachrichtenkopfes als auch des Inhalts zu beeinflussen. **DataContract** benutzt zur Definition einfache Klassen, deren Elementen diverse *Attribute*³ hinzugefügt werden. Darüber hinaus kann eine Nachricht auch durch eine Auflistung verschiedener Parameter definiert werden. In jedem Fall wird innerhalb der Xaml lediglich auf die Klassentypen verwiesen. Das Vorgehen entspricht den Verweisen auf primitive Typen in einer XSD (vgl. [Buk10, S.322 f.]). Datenmanipulationen innerhalb der Aktivitäten finden in WF durch die Verwendung von *Visual Basic*⁴ Ausdrücken statt. Mit ihnen können Daten gesetzt, Variablen verändert und Bedingungen ausgewertet werden (vgl. [Buk10, S.57 ff.]).

Um inhaltsbasierte Korrelation in WF zu nutzen, wird intern eine Variable des Typs **CorrelationHandle** verwendet, die die Korrelationsinformationen beinhaltet. An der ersten **Receive**–Aktivität wird durch die Eigenschaft *CorrelationInitializer* eine neue Korrelation initialisiert. Dies geschieht interaktiv durch Auswahl des **CorrelationHandles** und einer Variable der Nachricht, für die automatisch eine XPath–Query erstellt wird, in der sich der Korrelationswert befindet. In allen weiteren korrelierten Aktivitäten muss nun die Korrelationsvariable bei der Eigenschaft *CorrelatesWith* gewählt werden und wiederum eine Query auf den Wert in der zugehörigen Nachricht unter *CorrelatesOn*. Damit wird sichergestellt, dass jeweils bei dem Empfang einer Nachricht zur richtigen Instanz weitergeleitet oder eine neue Instanz gestartet wird (vgl.[Buk10, S.386 ff.]).

2.2 Workflow Engines

Um die mit einer Workflowsprache definierten Prozesse ausführen zu können, wird eine zugehörige Workflow Engine benötigt. Bei einer Workflow Engine handelt es sich um eine Software, die die Laufzeitumgebung für eine Prozessinstanz, basierend auf einer Prozessdefinition, bereitstellt. Dazu interpretiert oder kompiliert sie die mit einer Workflowsprache definierte Prozessdefinition zunächst in einen ausführbaren Prozess. Anschließend verwaltet die Engine den kompletten Lebenszyklus der Prozessinstanz, von der Instanzierung, über die Ausführung der orchestrierten Aktivitäten, bis hin zur erfolgreichen Beendigung

³Dahinter verbirgt sich eine .NET–spezifische Bezeichnung für Metadaten–Annotationen.

⁴Visual Basic ist eine proprietäre objektorientierte Programmiersprache innerhalb des .NET Frameworks.

oder dem Abbruch der Prozessinstanz (vgl. [WFM99, S.57]). Die beiden Unterkapitel stellen die beiden für die Workflowsprachen verwendeten Engines kurz vor.

2.2.1 Windows Workflow Foundation

WF ist fester Bestandteil des .NET Frameworks. Das .NET Framework ist Microsofts Software-Platform zur Entwicklung und Ausführung von Anwendungsprogrammen. Es besteht aus zwei Hauptkomponenten: Der *Common Language Runtime (CLR)* und der .NET Klassenbibliothek, die eine große Anzahl an vorgefertigten Komponenten zur Entwicklung von Anwendungen zur Verfügung stellt. Das Framework unterstützt unterschiedliche Sprachen, um Anwendungen innerhalb des Frameworks zu entwickeln. Alle diese Sprachen werden zunächst in eine Zwischensprache namens *Common Intermediate Language (CIL)* kompiliert. Die CLR kompiliert diesen Code zur Laufzeit mit einem Just-In-Time-Compiler in einen maschinenlesbaren Code. Dadurch ist es möglich sprachneutral Anwendungen zu entwickeln, die miteinander interagieren können (vgl. [Mic12j]). Innerhalb des Frameworks existieren neben diversen Basisklassen weitere Teilframeworks, die verschiedene Anwendungsgebiete kapseln. Darunter auch die *Windows Workflow Foundation*, die alle nötigen Komponenten für die Entwicklung von prozessbasierten Anwendungen enthält. Abbildung 1 zeigt die Komponenten und Zusammenhänge des Frameworks im Überblick.

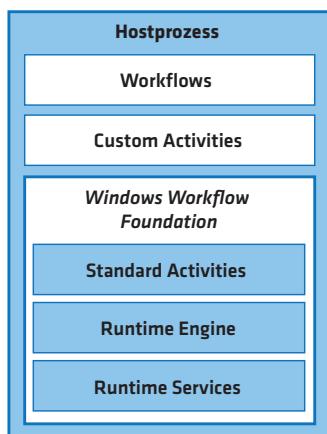


Abbildung 1: Übersicht der Windows Workflow Foundation Architektur [Mic07]

Das Framework beinhaltet die Abbildung des vorgestellten Prozessmodells auf Standardaktivitäten und eine zugehörige Workflow Engine zur Ausführung der Workflows. Auf Basis des Frameworks können weitere **CustomActivities** sowie eigene Workflows erstellt werden. Die Workflows werden von der Engine innerhalb eines Hostprozesses ausgeführt. Als Hostprozess kann dabei jede beliebige .NET Anwendung zum Einsatz kommen. Workflows können damit sowohl in Konsolenanwendungen als auch in Webanwendungen, beispielsweise als Webservice ausgeführt werden. Sämtliche Kommunikation über Webservices in WF wird über die WCF abgewickelt. WCF ist das serviceorientierte Kommunikationsframework für verteilte Anwendungen innerhalb des .NET-Frameworks. Jede Workflowinstanz wird von der Engine in genau einem Thread ausgeführt. Deshalb ist es nicht möglich, für die Standardaktivitäten eine echt parallele Ausführung zu erreichen. Alle Aktivitäten, deren Ausführung parallel geplant sind, werden pseudo-parallel innerhalb des Threads ausgeführt. Für die Entwicklung von .NET-Programmen vertreibt Microsoft die

Entwicklungsumgebung Visual Studio. Visual Studio enthält einen visuellen Workflow–Designer, mit dem alle nötigen Schritte der Modellierung und Konfiguration von Workflows durchgeführt werden können. Die meisten Einstellungen um einen Workflow auszuführen werden dabei von oder in der Entwicklungsumgebung konfiguriert. Für das Deployment als `WorkflowService` existiert zudem eine Konfigurationsdatei (`WEB.CONFIG`) in der bei Bedarf manuell detaillierte Einstellungen durchgeführt werden können. Listing 3 zeigt eine beispielhafte Konfigurationsdatei, die einen benutzerdefinierten Endpoint und ein zugehöriges Binding konfiguriert (vgl. [Mic12g])⁵.

Listing 3: WCF Konfigurationsdatei (nach [Mic12g])

```

1 <configuration>
2   <system.servicemodel>
3     <services>
4       <!-- Define the service endpoints -->
5       <service name="Tourist">
6         <endpoint binding="basicHttpBinding" contract="Tourist"
7           bindingConfiguration="customConfiguration"/>
8       </service>
9     </services>
10    <bindings>
11      <!-- Configure binding elements -->
12      <basicHttpBinding>
13        <binding name="customConfiguration"
14          closeTimeout="00:01:00" />
15      </basicHttpBinding>
16    </bindings>
17  </system.servicemodel>
18 </configuration>
```

2.2.2 Apache Orchestration Director Engine

Da BPEL im Gegensatz zu WF nur eine Sprache zur Beschreibung von Geschäftsprozessen ist, ohne zugehörige Entwicklungsumgebung und Workflow Engine, werden diese Komponenten zusätzlich benötigt. Hierfür wird eine Kombination der Open Source Anwendungen *Apache Orchestration Director Engine (Apache ODE)*⁶ als Laufzeitsystem und der Entwicklungsumgebung Eclipse⁷, zusammen mit dem BPEL–Designer–Plugin⁸, gewählt.

Apache ODE stellt das Laufzeitsystem für die mit BPEL erstellten Geschäftsprozesse. Es ist in Java implementiert und somit plattformübergreifend einsetzbar. Apache ODE unterstützt dabei sowohl die aktuelle BPEL Version 2.0 als auch die alte BPEL4WS 1.1 Spezifikation. Dabei werden jedoch noch nicht alle in der Spezifikation definierten Anforderungen vollständig erfüllt (siehe hierzu [Apa12b]). ODE beinhaltet alle nötigen Komponenten für die Ausführung, die Persistierung und die Kommunikation der Prozessinstanzen. Für die Kommunikation über Web Services nutzt ODE die Apache Axis 2 Engine. Apache ODE muss nicht zwingend in einem J2EE Application Server gehostet

⁵Mehr Informationen zu den vielfältigen Konfigurationsoptionen unter <http://msdn.microsoft.com/de-de/library/ms733932.aspx>

⁶Dokumentation und Download unter <http://ode.apache.org>

⁷Dokumentation und Download unter <http://www.eclipse.org/>

⁸Dokumentation und Download unter <http://www.eclipse.org/bpel/>

werden. Aufgrund der verwendeten Komponenten genügt ein Servlet Container, wie z.B. Apache Tomcat⁹ (vgl. [Apa12a]). Jedes Deployment in Apache ODE besteht aus einem Verzeichnis mit allen zugehörigen Artefakten. Als Minimalanforderung bestehen diese aus einer Prozessdefinition, der zugehörigen WSDL und möglicher XSDs. Damit der Prozess von Apache ODE auch veröffentlicht wird, ist zudem ein sog. *Deployment Descriptor* (DEPLOY.XML) nötig. Listing 4 zeigt einen Auschnitt einer solchen Beschreibung.

Listing 4: Deployment Descriptor Apache ODE

```

1 <deploy xmlns:ba="http://uniba.de">
2   <process name="ba:Guide">
3     <provide partnerLink="GuideService">
4       <service name="ba:Guide" port="GuidePort"/>
5     </provide>
6     <invoke partnerLink="TouristService">
7       <service name="ba:Tourist" port="TouristPort"/>
8     </invoke>
9   </process>
10 </deploy>
```

Das Wurzelement `<deploy>` enthält eine Liste aller veröffentlichten Prozesse. Jeder Prozess wird dabei mit seinem qualifizierten Namen identifiziert. Dabei müssen alle verwendeten `partnerLinks` auf konkrete Webservices abgebildet werden. Jeder `partnerLink` der in einer `<receive>`-Aktivität genutzt wurde muss ein entsprechendes `<provide>`-Element aufweisen, das auf einen Service und einen konkreten Endpoint (Attribut `port`) aus einer der zugehörigen WSDLs verweist. Das Gleiche gilt für alle `partnerLinks` aus `<invoke>`-Aktivitäten, es sei denn das Attribut `initializePartnerRole` wurde auf `false` gesetzt. Diese werden in `<invoke>`-Elementen definiert. Ein Endpunkt kann dabei jeweils nur einem `partnerLink` zugewiesen werden. Um den Prozess zu deployen, genügt es den Ordner mit allen Artefakten in den Unterordner WEB-INF/PROCESSES der ODE-Installation zu kopieren (vgl. [Apa12c]).

Um es vorweg zu nehmen, auch wenn Eclipse-basierte Entwicklungsumgebungen besonders bei Java-Entwicklern beliebt sind, empfiehlt es sich im Falle der BPEL-Entwicklung nicht auf das angebotene BPEL-Designer-Plugin zurückzugreifen. Während der Entwicklung der Fallbeispiele wurden viele schwer aufspürbare Probleme allein durch Fehler im Designer verursacht. Der Designer erstellt u.a. viele nutzlose Dummy-Variablen, die beim Entfernen nicht restlos gelöscht werden. Bei Variablentypänderungen werden zudem manchmal doppelte Einträge geschrieben, die schwer nachvollziehbare Fehlermeldungen in darauf bezogenen XPath-Queries erzeugen können, da die Variable anschließend an verschiedenen Stellen als unterschiedlicher Typ erkannt wird. Weiterhin verunsichern viele Warnungen bei der Auswertung von Queries oder Zuweisungen in Assign-Aktivitäten. Zudem existieren zwischen der Implementierung von Apache ODE und der BPEL-Spezifikation einige Unterschiede sowie nicht unterstützte Konstrukte (vgl. [Apa12b]). Beispielhaft sei hier das *Message Part*-Mapping (vgl. [OAS07, S.88 f.]) genannt, mit dem direkt Elementvariablen über ein virtuelles `assign` von oder zu einer SOAP-Nachricht zugewiesen werden können. Demnach müssen immer Nachrichtentypen als Variablen in Nachrichtenaktivitäten referenziert werden. Der Designer bedient hier die normale Spezifikation, was zu einem nicht ausführbaren Quellcode führt. Des Weiteren hält sich der Designer nicht an alle korrekten Reihenfolgebeziehungen innerhalb der Elemente der

⁹Dokumentation und Download unter <http://tomcat.apache.org>

BPEL-Spezifikation (siehe Kapitel 6.2.2). Bei einer Gesamtbetrachtung aktuell verfügbarer BPEL-Designer lässt sich generell eine absteigende Tendenz erkennen. Während der Eclipse-BPEL-Designer nach längerer Zeit gerade eine 1.0 Version mit beschriebener Qualität erreicht hat, hat NetBeans den SOA-Support und damit den BPEL-Designer nach Version 6.7 aus seinem Produkt entfernt¹⁰. Auch Active Endpoints hat den ActiveBPEL-Designer offensichtlich eingestellt. Als kommerzielles Produkt ist lediglich der Oracle JDeveloper 11g Release 2¹¹ auf dem Markt. Aus den genannten Gründen wurden die BPEL-Prozesse der Fallbeispiele deshalb größtenteils manuell nachbearbeitet.

¹⁰Aktuelle Version 7.1.2 unter <http://netbeans.org>

¹¹Die BPEL-Unterstützung ist als Teil der *SOA Suite* verfügbar. Dokumentation und Download unter <http://www.oracle.com/us/technologies/soa/soa-suite-066466.html>

3 Serviceinteraktion mit Petri-Netzen

Petri-Netze haben sich im Bereich der Modellierung von Geschäftsprozessen bewährt (vgl. [Van98, Vv04]). Gründe hierfür sind neben der intuitiven graphischen Darstellung, vor allem die Ausdrucksmächtigkeit und die formale Semantik der Petri-Netze. Petri-Netze besitzen die Fähigkeiten, alle Konstrukte heutiger Workflowsprachen zu modellieren. Weiterhin stehen eine Vielzahl von Analysetechniken zur Verfügung, um die Eigenschaften der Netze zu überprüfen (z.B. Deadlockfreiheit). Der fundierte mathematische Hintergrund erlaubt die Beweisführung über diese Eigenschaften (vgl.[Van98, S.24 f.]). Ziel der Arbeit ist es, die mit WF und BPEL modellierten Services in Petri-Netz-Repräsentationen zu transformieren und deren formalen Grundlagen für eine Integration zu nutzen. Für den Fall eines mit BPEL modellierten Services existiert bereits ein Compiler, *BPEL2oWFN*¹², um einen BPEL-Prozess in ein Petri-Netz zu übersetzen. Für WF soll im Zuge der Arbeit ebenso ein Prototyp implementiert werden. Im Folgenden werden deshalb, neben einer Definition des verwendeten Petri-Netz-Formalismus, grundlegende Konzepte der Serviceinteraktion auf Petri-Netz-Ebene vorgestellt. Besonderes Augenmerk liegt dabei auf der angestrebten Integration und Adaption von Services.

3.1 Petri-Netze

Bevor der in dieser Arbeit verwendete Modellierungsformalismus für oWFN präsentiert wird, sollen zunächst kurz die grundlegenden Formalismen der Petri-Netze vorgestellt werden.

Stellen/Transitions-Netze

Petri-Netze sind formale Systeme, die aus zwei Arten von Knoten, nämlich Stellen (Places) und Übergängen (Transitionen), sowie einer Flussrelation zwischen den Knoten bestehen. Die Flussrelation kann in eine Inputrelation, welche die Zuordnung von Stellen zu den Transitionen als Inputstellen vornimmt und eine entsprechende Outputrelation unterteilt werden. Grafisch wird eine Stelle durch einen Kreis und eine Transition durch eine Box dargestellt. Ein Zustand eines Petri-Netzes wird durch seine Markierung bestimmt, d.h. die Verteilung von Marken (Token) auf die Stellen des Netzes. Ein Token wird hierbei durch einen Punkt dargestellt. Das dynamische Verhalten eines Petri-Netzes wird durch seine Ausführung sichtbar. Ausgehend von der Anfangsmarkierung besteht die Ausführung eines Petri-Netzes durch das Schalten von zulässigen Übergängen. Ein Übergang ist zulässig, falls alle vorangehenden Inputstellen mindestens eine Marke enthalten. Beim Schalten wird jeweils eine Marke der Inputstellen entfernt und jeder Outputstelle eine neue hinzugefügt. Die Anzahl der benötigten bzw. der hinzuzufügenden Marken kann ggf. durch Kantenmarkierungen beeinflusst werden. Die Ausführung eines Petri-Netzes ist nicht deterministisch, jedoch wird grundsätzlich immer nur eine Transition geschalten. Ein Petri-Netz hat endlich viele Zustände, wenn es *k*-beschränkt ist, d.h. in keiner Stelle sammeln sich unter jeder erreichbaren Markierung mehr als *k* Token an (vgl. [VMSW09, S.15]).

¹²Eine kompilierte Version befindet sich auf der CD im Ordner SOFTWARE\BPEL2OWFN. Dokumentation und Download unter <http://download.gna.org/service-tech/bpel2owfn/>

Definition 1 (Petri-Netz) Ein Petri-Netz $N = [P, T, F, m_0]$ besteht aus

- zwei endlichen disjunkten Mengen P und T von Stellen und Übergängen,
- einer Flussrelation $F \subseteq (P \times T) \cup (T \times P)$ und
- einer initialen Markierung m_0 , wobei eine Markierung ein Mapping $m : P \rightarrow \mathbb{N}$ ist.

Open Workflow Nets

Bei Open Workflow Nets [MRS05] (im Folgenden kurz *Open Nets*) handelt es sich um eine Verfeinerung der Petri-Netze im Kontext der Servicemodellierung. Ein Service besteht aus einer Kontrollstruktur, die sein Verhalten beschreibt und einem Interface, um mit anderen Services kommunizieren zu können. Hierfür besteht ein Interface aus einer Menge an Input- und Output-Ports. Damit zwei Services miteinander interagieren können, müssen ein Input-Port des einen Services und ein Output-Port des anderen Services miteinander verbunden werden. Die verbundenen Ports formen dann einen Kommunikationskanal. Solche Services werden als Open Nets modelliert. Ein Open Net beinhaltet hierbei ein Petri-Netz, das die Kontrollstruktur des Services adäquat darstellen kann. Zusätzlich besitzt ein Open Net spezielle Interfacestellen für die Kommunikation und festgelegte finale Markierungen, in denen es erfolgreich terminieren kann. Das Service-Interface ist durch zwei disjunkte Mengen an Input- und Output-Stellen abgebildet. Hierbei korrespondiert jede Interfacestelle mit einem Port. Definition 2 zeigt die formale Beschreibung eines Open Nets (vgl. [VMSW09, S.17]).

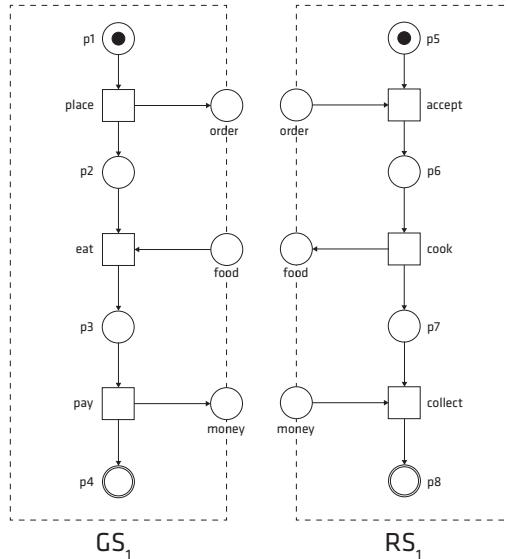
Definition 2 (Open Net) Ein Open Net $N = [P, T, F, I, O, m_0, \Omega]$ besteht aus einem Petri-Netz $[P, T, F, m_0]$ zusammen mit

- einem Interface $(I \cup O) \subseteq P$, definiert als zwei disjunkte Mengen I von Inputstellen und O von Outputstellen, sodass alle $p \in I$ keine Vorgängerelemente und alle $p \in O$ keine Nachfolgerelemente haben und
- einer Menge Ω von finalen Markierungen.

Weiterhin dürfen die Interfacestellen in der initialen und finalen Markierung nicht markiert sein, d.h. für alle Markierungen $m \in \Omega \cup \{m_0\}$ gilt $m(p) = 0$, für alle $p \in I \cup O$.

Grafisch wird ein Open Net wie ein Petri-Netz innerhalb eines gestrichelten Rahmens dargestellt. Die Interfacestellen werden auf diesem Rahmen abgebildet. Hierdurch wird die Trennung zwischen dem inneren Teilnetz und dem Interface des Netzes verdeutlicht. Die finalen Markierungen müssen separat definiert werden. Zur besseren Kenntlichkeit sind finale Stellen durch einen Doppelkreis visualisiert. Ein Open Net mit einem leeren Interface, d.h. keinerlei Interfacestellen, ist ein sog. *geschlossenes Netz*. Ein geschlossenes Netz kann benutzt werden, um beispielsweise eine Service-Choreographie zu modellieren (vgl. [VMSW09, S.17]).

Abbildung 2 zeigt zwei einfache Services. Das Open Net GS_1 modelliert einen Gast, der zunächst eine Bestellung aufgibt, Essen zu sich nimmt und anschließend bezahlt. Diese


 Abbildung 2: Gast–Service GS₁ und Restaurant–Service RS₁ [VMSW09, S.3]

drei Aktivitäten des Services sind als Transitionen modelliert und die Verbindung zu einem anderen Service über entsprechende Interfacestellen ($I = \{food\}$, $O = \{order, money\}$) vorgesehen. Als finale Markierung wird $\Omega = \{[p4]\}$ definiert. Das Open Net RS₁ zeigt einen Restaurant–Service, der zunächst eine Bestellung aufnimmt, Essen zubereitet und anschließend Geld kassiert. Dieser Service ist das offensichtliche Gegenstück zu GS₁.

3.2 Komposition

Die Kernidee einer SOA ist es, einzelne Services als Bausteine für die Erstellung von komplexen Services zu nutzen. Um dies zu erreichen, müssen Services komponiert werden, d.h. Input– und Output–Ports dieser Services miteinander verknüpft werden. Die Kommunikation zwischen den Services findet durch Nachrichtenaustausch über die resultierenden Kanäle statt. Die Komposition zweier Open Nets wird durch die paarweise Verschmelzung zweier gleichbezeichneter Input– und Outputstellen modelliert. Hierfür nimmt man an, dass alle Komponenten außer den Interfacestellen paarweise disjunkt sind. Dies kann einfach durch Umbenennung erreicht werden. Dem gegenüber überlappen die Interfacestellen absichtlich. Weiterhin ist es zweckmäßig zu erwarten, dass alle Kommunikation bilateral und gerichtet ist, d.h. für jede Interfacestelle $p \in I \cup O$ existiert nur ein Open Net, das in p sendet und das von p empfängt. Die Outputstelle des sendenden und die Inputstelle des empfangenden Netzes sind hierfür gleich benannt. Open Nets, die diese Eigenschaften erfüllen, werden als *interface-kompatibel* bezeichnet. Als Ergebnis einer Komposition von zwei Open Nets erhält man wiederum ein Open Net (vgl. [VMSW09, S.18]). Definition 3 zeigt die formale Beschreibung einer Komposition zweier Netze.

Definition 3 (Komposition) Es seien N_1 und N_2 zwei interface-kompatible Open Nets. Die Komposition $N = N_1 \otimes N_2$ ergibt ein Open Net mit den folgenden Bestandteilen:

- $P = P_1 \cup P_2$,
- $T = T_1 \cup T_2$,

- $F = F_1 \cup F_2$,
- $I = (I_1 \cup I_2) \setminus (O_1 \cup O_2)$,
- $O = (O_1 \cup O_2) \setminus (I_1 \cup I_2)$,
- $m_0 = m_{01} \otimes m_{02}$ und
- $\Omega = \{m_1 \otimes m_2 \mid m_1 \in \Omega_1, m_2 \in \Omega_2\}$.

Kompatibilität. Die Open Nets in Abb.2 sind interface-kompatible Netze. Sie können komponiert werden, indem gleich beschriftete Interfacestellen zusammengefügt werden. Jede Komposition resultiert in diesem Falle in ein neues geschlossenes Open Net. Die resultierende Komposition einer Menge an Services soll weiterhin in sich kompatibel sein. Die Komposition der beiden Services hat nur einen möglichen Ausführungspfad: *place* \rightarrow *accept* \rightarrow *cook* \rightarrow *eat* \rightarrow *pay* \rightarrow *collect*. Es ist offensichtlich, dass die beiden Services kompatibel sind, ohne bereits eine präzise Definition im Sinn zu haben. Offenkundig existiert keine eindeutige Definition von Kompatibilität. Die Minimalanforderung jedoch ist das Nichvorhandensein von *Deadlocks* in einem Service. Ein Deadlock ist ein erreichbarer, nicht finaler Zustand m in N , in dem das Open Net stecken bleibt, d.h. keine Transition m aktiviert bzw. aktivierbar ist. Ein strikteres Kriterium ist die Möglichkeit, dass ein Service aus jedem erreichbaren Zustand terminieren kann. Dieses Kriterium schließt neben Deadlocks auch noch sog. *Livelocks* aus. Ein Livelock ist eine Menge an erreichbaren Zuständen eines Services, aus denen weder ein Deadlock noch ein finaler Zustand erreichbar ist, d.h. das Netz pendelt immer zwischen mehreren Zuständen hin und her. Für die weitere Betrachtung ist ein geschlossenes Netz kompatibel, falls es deadlockfrei ist (vgl. [VMSW09, S.19]).

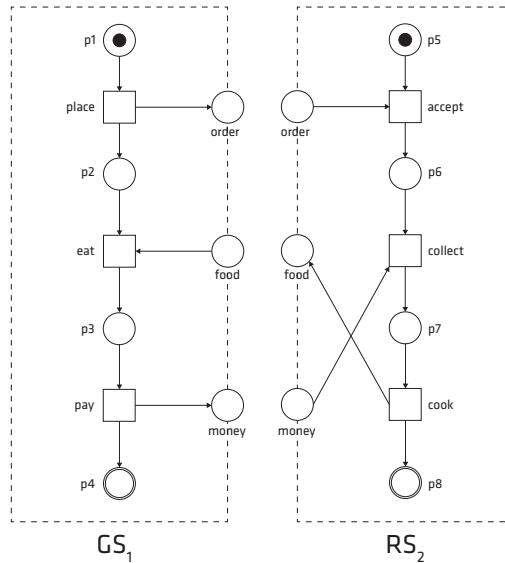
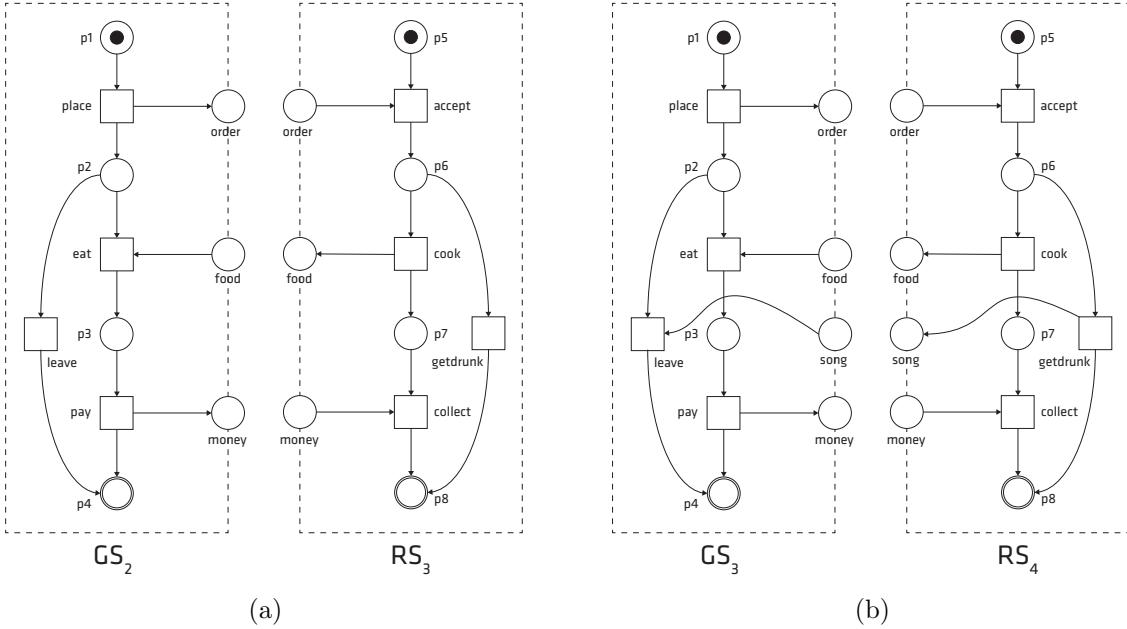


Abbildung 3: Gast–Service GS_1 und Restaurant–Service RS_2 [VMSW09, S.3]

Im Folgenden sei $\Omega = \{[p4, p8]\}$ für alle Kompositionen in den Abb.2–4. Die Komposition $GS_1 \otimes RS_1$ in Abb.2 ist deadlockfrei und wie schon vermutet kompatibel. Abbildung 3 zeigt den Service GS_1 und einen neuen Restaurant–Service RS_2 , der jedoch weniger zuvorkommend ist und eine Bezahlung verlangt, bevor das Essen zubereitet wird. Die Komposition


 Abbildung 4: Gast- und Restaurant-Services $GS_2 \otimes RS_3$ und $GS_3 \otimes RS_4$ [VMSW09, S.4]

$GS_1 \otimes RS_2$ läuft immer in einen Deadlock, nämlich nachdem eine Bestellung auf- und angenommen wurde, sind beide Services blockiert und warten aufeinander ($[p2, p6]$). Diese Services sind offensichtlich nicht miteinander kompatibel. Dies führt zum Konzept der Strategie eines Netzes. Definition 4 zeigt das Konzept, das die Verhaltenskorrektheit zweier Netze definiert. Die zugehörige Menge $Strat(N)$ beinhaltet alle kompatiblen Netze des Netzes N . Falls ein Netz nicht steuerbar ist, ist es fundamental schlecht konzipiert, da es mit keinem anderen Netz ordnungsgemäß interagieren kann. In unseren Beispielen ist GS_1 eine Strategie für RS_1 und umgekehrt.

Definition 4 (Strategie, Steuerbarkeit) Seien M, N zwei Open Nets sodass $I_M = O_N$ und $O_M = I_N$. Dann ist M eine Strategie für N , gdw. $M \otimes N$ deadlockfrei ist. Mit $Strat(N)$ wird die Menge aller Strategien für N bezeichnet. N ist steuerbar, gdw. die Menge seiner Strategien nicht leer ist.

Abbildung 4a zeigt ein weiteres Beispiel. Dieses Mal kann der Gast ohne zu Essen und zu Zahlen das Restaurant verlassen. Zusätzlich ist der Koch Alkoholiker und könnte sich betrinken, statt das Essen zuzubereiten. Die Komposition $GS_2 \otimes RS_3$ beinhaltet offensichtlich einen Deadlock ($[p4, food, p7]$). Es kann passieren, dass der Kunde gegangen ist, obwohl der Koch Essen zubereitet. Es stellt sich die Frage welche Services mit RS_3 respektive GS_2 kompatibel sein könnten, da ihre internen Entscheidungen für die Umwelt unklar bleiben. Jedoch sind auch diese beiden Netze steuerbar. Dies mag auf den ersten Blick überraschend sein, jedoch wäre ein Restaurant-Service, der nur die Bestellung aufnimmt und dann beendet eine Strategie für GS_2 , die damit den Kunden zum Gehen zwingt. Einer Strategie für RS_3 muss bewusst sein, dass nach der Bestellung der Koch betrunken werden könnte und in diesem Falle kein Essen serviert werden würde. Nichtsdestotrotz bekommt der Kunde darüber keine Auskunft und muss deshalb im Restaurant bleiben, falls Essen serviert wird. Dies könnte z.B. vom Open Net GS_1 mit den finalen Markierungen $\Omega = \{[p2], [p4]\}$ modelliert werden, d.h. nach einer Bestellung muss der Gast nicht essen, aber wenn serviert wird, isst und zahlt er (vgl. [VMSW09, S.20]). Abbildung 4b zeigt

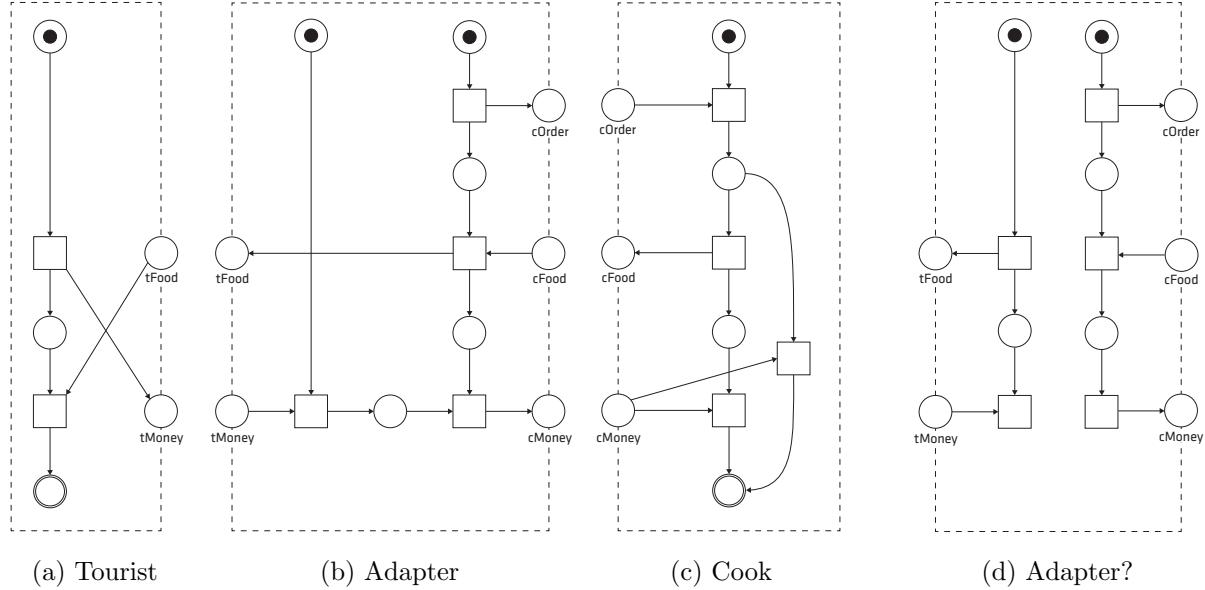


Abbildung 5: Integration mit einem Adapter-Service [VMSW09, S.39]

zwei angepasste Versionen der Services. In der neuen Choreographie beginnt der Koch zu singen wenn er sich betrinkt und signalisiert damit dem Kunden zu gehen. Diese beiden Services beinhalten keine nicht beobachtbaren internen Entscheidungen und sind wieder kompatibel.

3.3 Integration und Adaption

Serviceorientiertes Computing zielt darauf ab komplexe Services durch die Komposition von weniger komplexen Services zu erreichen. Da Services häufig unabhängig voneinander entwickelt werden, kann sich bei einer Komposition herausstellen, dass sie inkompatibel sind. Wenn solche Services integriert werden, können einige typische Probleme auftreten. Einige davon sind:

- Namen der Nachrichtentypen
- Kodierung von gleichen Nachrichtentypen
- Semantik von gleichen Nachrichtentypen
- Reihenfolge in der Nachrichten erwartet oder gesendet werden

Abbildung 5 zeigt ein Beispiel hierfür. Der Tourist modelliert in Abb.5a betritt ein Restaurant in einem fremden Land. Er bemerkt ein spezielles Angebot an der Tür, spricht aber die lokale Sprache nicht. Deshalb legt er lediglich die erforderliche Gebühr auf den Tisch und wartet auf das Essen. Der ansässige Koch (modelliert durch Abb.5c) besteht jedoch auf eine Bestellung, bevor er eine Mahlzeit zubereitet. Zudem hört er sofort mit dem Kochen auf, wenn er Geld bekommt, bevor er das Essen serviert hat.

Es ist klar, dass die Open Nets Abb.5a und Abb.5c verschiedene Mengen von Interfacestellen haben, auch wenn man die Präfixe zunächst ignoriert. Wenn man versucht die

offensichtlichen Kombinationen der Interfacestellen zu verbinden und die anderen Interfacestellen zu verstecken, d.h. intern machen, dann enthält das Resultat einen Deadlock. Der Koch wartet auf eine Bestellung, während der Tourist Geld bezahlt und auf das Essen wartet. Der Fokus soll in diesem Kapitel auf dem letzten Typen der genannten Inkompatibilität liegen, welche man Verhaltensinkompatibilität nennt. Aus Gründen der Einfachheit nehmen wir an, dass der Name eines Ports mit dem Namen des Nachrichtentypen der über den Kanal übertragen wird übereinstimmt. Wenn die Services inkompatibel sind, gibt es einige Optionen:

1. Ersetze einige der Services mit gleichen Services die kompatibel sind
2. Ändere die Implementierung einiger Services
3. Führe einen Adapter-Service ein, der die Inkompatibilitäten überbrückt

Im konkreten Beispiel bedeutet dies, dass der Tourist 1) zu einem anderen Restaurant mit touristenfreundlichem Personal geht, 2) einen Sprach- und Kulturkurs belegt oder 3) einen Tourguide anheuert. Dieses Kapitel soll sich mit der Situation beschäftigen, in der die Services bereits ausgewählt worden sind und ihre Implementierung nicht mehr geändert werden kann. In diesem Falle sind Adapter die offensichtlichste Lösung. Um Adapter als einen zusätzlichen Service zwischen den gegebenen Services zu diskutieren, wird angenommen, dass die Interfaces der Services disjunkt sind, was durch Umbenennung erreicht werden kann. Hier durch „t“- und „c“-Präfixe. Der Rest des Kapitels diskutiert die Bestandteile einer Adapterspezifikation und zeigt anschließend wie sie dazu genutzt werden können, um automatisch einen Adapter zu generieren (vgl. [VMSW09, S.34 f.]).

3.3.1 Adapterspezifikation

Verhaltensinkompatibilitäten manifestieren sich typischerweise in Form von Deadlocks im komponierten System. Deswegen ist der erste Bestandteil der Spezifikation eine Verhaltenseigenschaft, in diesem Falle die Deadlockfreiheit des Systems. Um diese überprüfen zu können, muss die Adapterspezifikation auch die Modelle der zu komponierenden Services beinhalten. Neben dem Adapter in Abb.5b erreicht auch der Adapter in Abb.5d diese Eigenschaft, aber ist dies ein korrekter Adapter? Das Beispiel zeigt, dass die Spezifikation bis jetzt Adapter erlaubt, die die Komposition zweier unverbundener Komponenten A_1 und A_2 ist, sodass beide $N_1 \otimes A_1$ und $N_2 \otimes A_2$ deadlockfrei sind, aber unabhängig voneinander. Dies erlaubt es Adaptern willkürlich Nachrichten zu erstellen oder zu löschen, einschließlich realer Güter wie in diesem Falle Nahrung und Geld, was nicht sehr realistisch ist. Deswegen wird eine zusätzliche Anforderung an die internen Bestandteile des Adapters gebraucht, sodass dieser implementiert werden kann. Zu diesem Zweck wird die Adapterspezifikation um die Menge der *elementaren Aktivitäten*, die vom Adapter benutzt werden können erweitert (vgl. [VMSW09, S.35 f.]). Dadurch besteht der Adapter aus den folgenden drei Teilen:

- Modelle der zu komponierenden Services
- Verhaltenseigenschaft, die vom komponierten System erreicht werden soll
- Elementare Aktivitäten für den Adapter

Elementary Adapter Activities. Die meisten Ansätze stimmen hierbei überein, dass die Aktivitäten eines Adapters folgende Aktivitäten einschließen sollten:

- *Create*: Möglich für einfache Kontrollnachrichten und Nachrichten mit einem Standardwert, jedoch nicht für Nachrichten mit wichtigen Daten wie Passwörtern oder persönlichen Daten eines Nutzers.
- *Copy*: Möglich für die meisten elektronischen Nachrichten, ausgenommen einmaliiger Transaktionsnummern, jedoch nicht geeignet für Nachrichten die reale Güter repräsentieren.
- *Delete*: Möglich für die meisten elektronischen Daten, jedoch ungeeignet für reale Güter.
- *Transform/ Split/ Merge*: Möglich falls die Transformationsroutine bekannt ist, z.B. Umwandlung einer Postleitzahl in einen Ortsnamen.

Basierend auf diesen Beispielaktivitäten wird klar, dass die Anwendung einer Aktivität auf eine bestimmte Nachricht stark von der semantischen Betrachtung der Nachrichtentypen abhängt. Deshalb müssen die möglichen Aktivitäten eines Adapters für die jeweiligen Nachrichtentypen definiert werden. Die Spezifikation der Möglichkeiten eines Adapters erfolgt durch die sog. *Specification of the Elementary Activities (SEA)*. Für eine gegebene Menge an Nachrichtentypen MT , ist die SEA eine Menge an Transformationsregeln für diese Nachrichtentypen. Die Menge MT besteht zumindest aus den Nachrichtentypen der Interfacestellen der gegebenen Services, kann aber auch zusätzliche Nachrichtentypen beinhalten (vgl. [VMSW09, S.36]).

Definition 5 (Specification of the Elementary Activities)

Für eine gegebene Menge an Nachrichtentypen MT ist eine SEA eine Menge von Transformationsregeln der Form

$$X \xrightarrow{Z} Y$$

wobei X und Y Multimengen über der Menge MT sind und Z eine totale Funktion von Nachrichten des Typs X zu Nachrichten des Typs Y .

Solche Regeln formalisieren, dass durch die Transformation Z eine Nachricht vom Typ X konsumiert und eine Nachricht vom Typ Y produziert wird. Nachdem der Adapter eine Nachricht erhalten hat, kann er einige Transformationen intern auf den Nachrichten ausführen, bevor er diese weitersendet. Der Bau eines Adapters läuft dann darauf hinaus diese Regeln in der richtigen Reihenfolge anzuwenden und Nachrichten im richtigen Moment zu senden und zu empfangen (vgl. [VMSW09, S.37]).

Für das Beispielproblem in Abb.5 findet sich eine mögliche SEA in Tabelle 1. Die Regeln ermöglichen es einem Adapter die fehlende Bestellung durchzuführen und anschließend die weiteren Nachrichten jeweils weiterzuleiten.

Tabelle 1: SEA des Restaurant-Adapters [VMSW09, S.32]

	\longleftarrow	cOrder
cFood	\longleftarrow	tFood
tMoney	\longleftarrow	cMoney

Der Adapter aus Abb.5d missachtet hierbei die letzten zwei Regeln und wäre nicht gültig, während der Adapter in Abb.5b sich konform verhält. Die Regeln können für die Nutzung durch die Tools in einer einfachen Textdatei zeilenweise in folgender Form spezifiziert werden:

cFood \rightarrow tFood;

3.3.2 Adaptersynthese

Ein geschickter Ansatz [GMW08] zur Generierung eines Adapters ist zunächst die SEA in ein Open Net zu transformieren, das Teil des Adapters ist. Diesen Teil des Adapters nennt man *Engine* und den Rest des Adapters *Controller*. Dieser Ansatz erzeugt einen zweiteiligen Adapter, der die Daten und den Kontrollfluss trennt. Die Engine E ist ein Open Net, das alle elementaren Aktivitäten der SEA encodiert. Es hat ein Interface mit den zu verbindenden Open Nets N_1 und N_2 und kann deshalb sicherstellen, dass alle ausgehenden Nachrichten nur durch gültige Regeln aus den eingehenden Nachrichten erstellt werden. Zudem hat die Engine ein zusätzliches Interface zu einem Controller C . Dieses Interface erlaubt dem Controller zu entscheiden, in welcher Reihenfolge die elementaren Aktivitäten durchgeführt werden. Abb.6 zeigt eine schematische Repräsentation dieser Struktur (vgl. [VMSW09, S.38]).

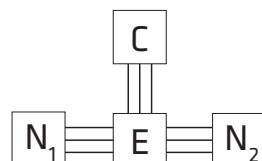


Abbildung 6: Konzeptuelle Adapterstruktur [VMSW09, S.39]

Auf diesem konzeptuellen Level besteht der Adaptersyntheseprozess aus den folgenden Schritten, bei gegebenen Open Nets N_1 und N_2 und einer SEA:

1. Generiere eine Engine E aus der SEA und dem Interface des Open Nets $N_1 \otimes N_2$
2. Synthetisiere einen Controller C als eine Strategie für das Open Net $(N_1 \otimes N_2) \otimes E$
3. Setze die Engine E und den Controller C zum finalen Adapter $A = E \otimes C$ zusammen

Im Folgenden wird der Prozess kurz erläutert. Aufgrund der Platzrestriktion sei für eine genauere Betrachtung auf [GMW08] hingewiesen. Es soll nun gezeigt werden, wie eine SEA als eine Engine durch ein Open Net encodiert werden kann. Zum besseren Verständnis zeigt Abb.7b die Engine des Adapters aus dem Beispiel des Kapitels.

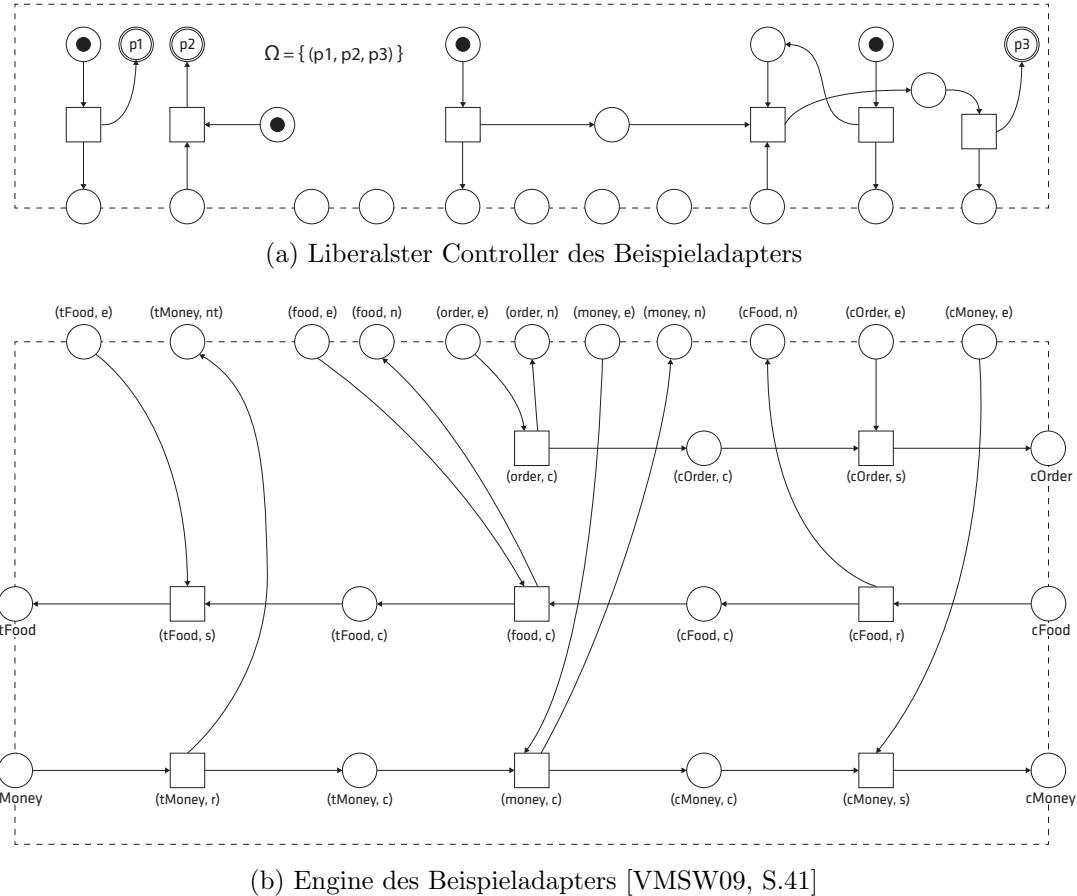


Abbildung 7: Controller und Engine des Beispieladapters

Das Interface der Engine besteht aus entsprechenden Input- und Outputstellen zu den gegebenen Services *Tourist* und *Cook* und einem Interface für die Interaktion mit dem Controller. Für jeden Nachrichtentypen existieren zwischen seiner Inputstelle und der Outputstelle drei verschiedene Transitionen. Zunächst wird die Nachricht empfangen und eine Benachrichtigung an den Controller weitergegeben. Danach findet nach Freigabe des Controllers die eigentliche Transformation nach entsprechenden Regeln der SEA statt. Als letztes wird die transformierte oder erzeugte Nachricht an die Outputstelle gesendet. Für jede eingehende Transition existiert eine Outputstelle innerhalb des Controllerinterfaces, um den Empfang der Nachricht zu melden und für jede sendene Transition eine Inputstelle, um diese zu aktivieren. Für die internen Regeln existiert sowohl eine aktivernde Inputstelle, als auch eine benachrichtigende Outputstelle. Jede Transformation folgt entweder den Regeln der SEA oder kopiert lediglich Token von Stellen des gleichen Nachrichtentyps. Die Interfacestellen zum Controller regeln nur die Abfolge in der die Transitionen geschaltet werden können. Damit garantiert die Engine, dass der generierte Adapter der SEA folgt, unabhängig von den Spezifika des Controllers.

Für das resultierende Open Net $(N_1 \otimes N_2) \otimes E$ existieren mehrere potentielle Strategien und jede Strategie kann als ein Controller für den Adapter genutzt werden. Im Allgemeinen unterscheiden sich die Strategien vor allem in Größe und Laufzeitverhalten. Eine besonders interessante Strategie stellt hier wiederum die liberalste (*most-permissive*) Strategie dar, die alle möglichen Strategien beinhaltet. Dadurch entstehen die geringsten Beschränkungen für das Interface des Controllers. Ein Nachteil ist jedoch erwartungsgemäß die Größe dieser Strategie. Auf der anderen Seite gibt es meist viele Strategien, die kleiner

sind, jedoch restringieren diese häufig die Interaktion mit den gegebenen Open Nets und beschränken die Nebenläufigkeit der Aktivitäten. Deshalb gilt es eine Abwägung zwischen der Komplexität der Adapter-Synthese und der Qualität (Laufzeitverhalten, Größe) des resultierenden Adapters je nach vorgesehenen Anwendungsfall zu treffen (vgl. [VMSW09, 39-42]). Für Details des Algorithmus der Controllersynthese wird auf [Wol09] verwiesen. Das Ergebnis des liberalsten Controllers ist in Abb.7a dargestellt. Schlussendlich wird der Controller und die Engine zum endgültigen Adapter zusammengesetzt. Wie zu erkennen ist, benutzt der Controller nicht alle Regeln der Engine. Diese unbenutzten Teile des Netzes werden durch eine anschließende Petri-Netz-Reduktion eliminiert. Da bei dieser Reduktion auch Transitionssequenzen zusammengefügt werden, kann das Interface zwischen Engine und Controller im finalen Adapter unkenntlich werden (vgl. [GMW10, S.9]).

Die vorgestellte Adaptersynthese kann mit dem Tool *Marlene*¹³ automatisiert durchgeführt werden. Als Input werden die Modelle der gegebenen Services als Open Nets und eine Regeldatei mit der SEA benötigt.

¹³Eine kompilierte Version befindet sich auf der CD im Ordner SOFTWARE\MARLENE. Dokumentation und Download unter <http://download.gna.org/service-tech/marlene/>.

4 Patterns

Um Petri–Netze als formale Grundlage zu nutzen und WF in die vorhandene Werkzeugkette einzugliedern, sollen WF–Prozesse in äquivalente Petri–Netze transformiert werden. Wie in BPEL wird in WF ein Prozess erstellt, indem verschiedene Sprachkonstrukte komponiert werden. Um einen Prozess zu transformieren, werden deshalb die verschiedenen Konstrukte separat in eine Petri–Netz–Repräsentation umgewandelt und anschließend wieder zu einem Gesamtprozess zusammengesetzt. Solch eine einzelne Petri–Netz–Repräsentation formt das *Pattern* der entsprechenden Aktivität. Alle Patterns besitzen ein einheitliches Petri–Netz–Interface, um sie mit anderen Patterns verbinden zu können. Patterns, die strukturierte Aktivitäten von WF abbilden, können zudem eine entsprechende Anzahl von inneren Aktivitäten beinhalten, genauso wie die äquivalente WF–Aktivität (vgl. [Buk10, S.5], [Loh07, S.3 f.]).

Aktivitätslebenszyklus

Um das gemeinsame Interface aller Aktivitäten abzuleiten, hilft es den Lebenszyklus der Aktivitäten zu betrachten. Nachdem eine Instanz einer Aktivität zur Ausführung vorbereitet wurde (*Initialized*–Zustand), startet die Aktivität mit dem *Executing*–Zustand (vgl. [Mic12o]). Treten keine Ausnahmen auf, so verbleibt die Instanz in diesem Zustand, bis die Ausführung aller untergeordneten Aktivitäten beendet wurde. Nachdem alle weiteren, noch ausstehenden Arbeitsvorgänge abgeschlossen sind, geht die Instanz in den *Closed*–Zustand über. Übergeordnete Elemente einer Aktivitätsinstanz können zur Laufzeit den Abbruch einer untergeordneten Aktivität anfordern. Wenn die Aktivität abgebrochen werden kann, wird sie mit dem *Canceled*–Zustand abgeschlossen. Wird während der Ausführung einer Aktivität eine Ausnahme ausgelöst, versetzt die Laufzeitumgebung die Aktivität in den *Faulted*–Zustand (vgl. [Mic12b]). Abbildung 8 zeigt das aus dem Lebenszyklus abgeleitete Petri–Netz–Interface. Neben dem initialen Zustand *initialized*, den finalen Zuständen *closed*, *canceled* und *faulted*, existiert eine weitere Interfacestelle *cancel*, die es einem übergeordneten Element erlaubt, den Abbruch der Aktivität zu initiieren. Der Zustand *executing* wird durch das gesamte innere Netz repräsentiert. Die Interfacestellen für den Gutfall, d.h. den fehlerfreien Ablauf einer Aktivität, sind durch eine stärkere Umrandung gekennzeichnet. Die im Folgenden dargestellten Patterns betrachten jeweils nur den Gutfall der Aktivitäten. Dies hat mehrere Gründe: Aufgrund der Zeitrestriktion können nicht alle WF–Aktivitäten im Rahmen der Arbeit transformiert und im Prototyp implementiert werden. Es erscheint daher sinnvoll zunächst auf die Gruppe der Aktivitäten zu verzichten, die mit Fehler– und Abbruchbehandlung in Verbindung stehen und sich auf die Basisfälle zu beschränken. Eine Darstellung der gesamten Netze für die betrachteten Aktivitäten wäre ohne diese kompensierenden Aktivitäten überflüssig. Auch das Tool BPEL2oWFN, das für die BPEL–Transformation genutzt werden soll, implementiert zudem nicht für alle Konstrukte vollständige Netze, weswegen die Beschränkung auf den Gutfall zu einem gemeinsamen Nenner in der Abstraktion der Netze führt. Eine Auflistung der nicht betrachteten Aktivitäten, inklusive kurzer Begründungen, findet sich in Kapitel 4.4.

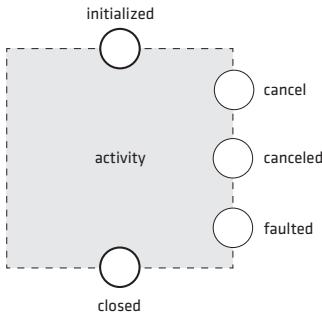


Abbildung 8: Interface-Pattern der Aktivitäten

Daten- und Zeitabstraktion

Die gezeigten Patterns abstrahieren alle Daten- und Zeitaspekte und stellen primär den Kontrollfluss der Aktivitäten dar. Dies ist nötig, da Variablen in WF im Allgemeinen einen unendlich großen Definitionsbereich aufweisen. Wenn man die Prozesse in High-Level-Netze mit Datenaspekten übersetzen würde, so könnten diese eine unendliche Zustandsmenge besitzen, was die anschließende Analyse unmöglich machen würde (vgl. [Loh07, S.28 f.]). Nachrichten und Variablen sowie deren Inhalt sind deshalb als nicht unterscheidbare Token modelliert. Datenabhängige Entscheidungen, wie z.B. Schleifenwiederholungen in `DoWhile`-Aktivitäten, sind demnach nichtdeterministische Entscheidungen. Listing 5 zeigt eine `DoWhile`-Beispielaktivität. Die Schleife wird über den Empfang einer Nachricht gesteuert. Der Inhalt der Nachricht wird in der Variable `choiceVar` gespeichert und zur Evaluation der Schleifenbedingung genutzt. Da die Nachrichteninhalte im Petri-Netz nicht unterscheidbar sind, hat das Netz jeweils die freie Wahl die Schleife zu wiederholen oder zu beenden.

Listing 5: Datenabstraktion anhand einer `DoWhile`-Aktivität

```

1 <DoWhile>
2   <DoWhile.Variables>
3     <Variable x>TypeArguments="x:Boolean" Name="choiceVar" />
4   </DoWhile.Variables>
5   <DoWhile.Condition>[choiceVar = True]</DoWhile.Condition>
6   <Receive OperationName="GetChoice">
7     <ReceiveMessageContent>
8       <OutArgument x>TypeArguments="x:Boolean">
9         [choiceVar]
10        </OutArgument>
11      </ReceiveMessageContent>
12    </Receive>
13 </DoWhile>
  
```

Leider wird durch dieses Verhalten die Steuerbarkeit des Netzes nur schwach erhalten, d.h. wenn das Petri-Netz steuerbar ist, wäre auch ein High-Level-Netz steuerbar und somit auch der ursprüngliche WF-Prozess. Diese Aussage gilt jedoch nicht automatisch in der Umkehrung. Ein High-Level-Netz bzw. ein WF-Prozess kann aufgrund der Daten, anhand deren die Entscheidungen getroffen werden, steuerbar sein, das transformierte Petri-Netz jedoch nicht, da die in diesem Fall für den Kontrollfluss relevanten Datenaspekte verloren gehen. Die gezeigte `DoWhile`-Schleife könnte beispielsweise jederzeit durch den Empfang eines falschen Wahrheitswertes von einem Partner abgebrochen werden. Im Petri-Netz ist

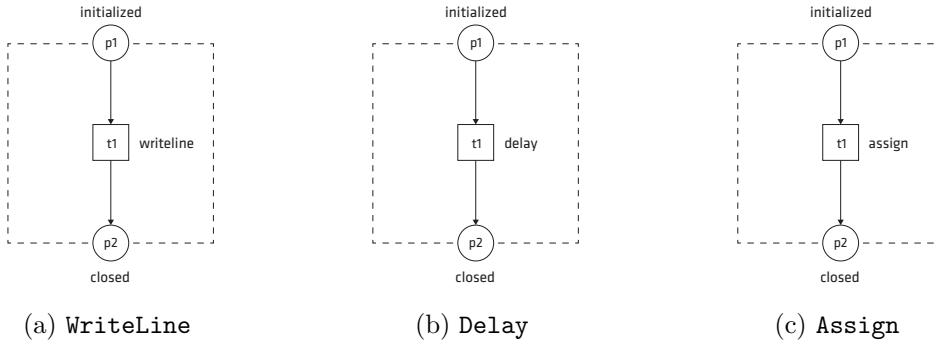


Abbildung 9: Patterns der primitiven Aktivitäten

dieser Wert jedoch nur ein einheitliches Token, weswegen der Aufrufende sich nicht sicher sein kann ob das Netz terminiert oder erneut die Schleife beginnt. Diese Problematik hat starke Auswirkungen auf die reale Anwendung, was konkret auch im Fallbeispiel in Kapitel 6.2.3 zu sehen ist. Die Beschränkungen der Betrachtung decken sich mit dem für BPEL2oWFN angewendeten Ansatz, was die nötige Grundlage für eine gemeinsame Integration schafft (vgl. [Loh07, S.29]).

4.1 Sequentielle Workflows

Dieses Kapitel zeigt zunächst alle transformierten Patterns für primitive und strukturierte Aktivitäten sequentieller Workflows. Sequentielle Workflows führen die enthaltenen Aktivitäten in der Reihenfolge ihrer Deklaration aus. WF unterstützt neben sequentiellen Workflows auch die beiden anderen graphbasierten Modellierungsarten Flowchart und State Machine (vgl. [Buk10, S.163], [Mic12i]). Primitive Aktivitäten führen in sich abgeschlossene Operationen durch, während strukturierte Aktivitäten (auch *Composites* genannt) weitere Aktivitäten beinhalten und den Kontrollfluss des Workflows beeinflussen.

4.1.1 Primitive Aktivitäten

WriteLine

Die **WriteLine**-Aktivität schreibt ein Literal an die Konsolenausgabe. Optional kann die Ausgabe an einen beliebigen anderen **TextWriter** weitergeleitet werden (vgl. [Buk10, S.108]). Abbildung 9a zeigt das Pattern der Aktivität. Da die Aktivität lediglich eine Textausgabe durchführt, besteht sie nur aus einer ausführenden Transaktion nach der die Aktivität terminiert.

Delay

Mit der **Delay**-Aktivität kann ein Timer innerhalb eines Workflows gesetzt werden, um dessen Ablauf zu verzögern. Wenn der Timer endet, wird die Ausführung des Workflows mit den nachfolgenden Aktivitäten fortgesetzt (vgl. [Buk10, S.107]). Da in der Arbeit auf

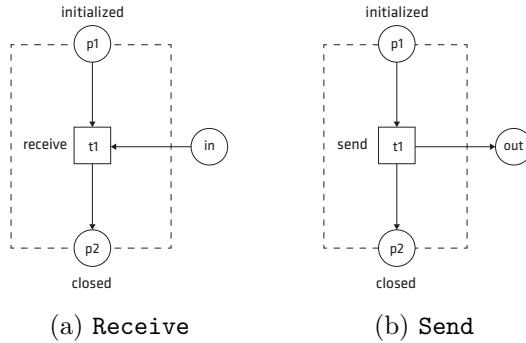


Abbildung 10: Patterns der Nachrichtenaktivitäten

die Modellierung der Zeitaspekte verzichtet wird, führt das entsprechende Petri–Netz–Pattern wiederum lediglich eine Transaktion durch, nach der die Aktivität beendet ist (siehe Abb.9b).

Assign

Die **Assign**-Aktivitt erlaubt es einer Variable oder einem Argument einen neuen Wert zuzuweisen. Dies kann sowohl der Wert einer anderen Variable, als auch eine direkte Zuweisung eines neuen Wertes sein (vgl. [Buk10, S.107]). Fr den Gutfall knnen die einzelnen Kopieroperationen auf eine Transition reduziert werden (siehe Abb.9c).

Receive und ReceiveReply

Die Aktivitäten `Receive`, `ReceiveReply`, `Send` und `SendReply` werden zur Interaktion mit anderen Anwendungen über die WCF genutzt. Die Patterns der Nachrichtenaktivitäten sind in Abbildung 10 zu sehen. Während die `Receive`-Aktivität den Empfang einer einzelnen Nachricht modelliert, kommt die `ReceiveReply`-Aktivität ausschließlich in Kombination mit einer korrelierten `Send`-Aktivität zum Einsatz. Die beiden Aktivitäten implementieren hierbei einen Request/Response-Nachrichtenaustausch (`SendAndReceiveReply`), bei dem `ReceiveReply` den Empfang der Antwortnachricht abbildet (vgl. [Buk10, S.106]). Da die Korrelation nicht modelliert wird, entspricht das Petri-Netz dem der `Receive`-Aktivität (siehe Abb.10a).

Send und SendReply

Die **Send**-Aktivität wird benutzt, um eine Nachricht an einen Service zu senden. Es handelt sich wiederum um eine einzelne Aktivität, die die Anfrage initiiert, aber keine Logik enthält, welche auf eine Antwort des Services wartet. Um auf eine empfangene Nachricht eine korrelierte Antwortnachricht zu senden, existiert die Kombination einer **Receive**- und einer **SendReply**-Aktivität (**ReceiveAndSendReply**). Die Petri-Netze der **Send**- und **SendReply**-Aktivitäten sind äquivalent und in Abb.10b abgebildet.

4.1.2 Strukturierte Aktivitäten

Sequence

Bei der **Sequence**-Aktivität handelt es sich um eine einfache strukturierte Aktivität, deren Hauptaufgabe es ist, die enthaltenen Aktivitäten unter Beachtung der Definitionsreihenfolge hintereinander auszuführen. Abb.11 zeigt das abgeleitete Pattern. Eine **Sequence** kann beliebig viele verschiedene Aktivitäten beinhalten. Per Definition ist die Endstelle einer Aktivität gleichzeitig die initiale Stelle der nächsten Aktivität (vgl. [Buk10, S.74, 104]).

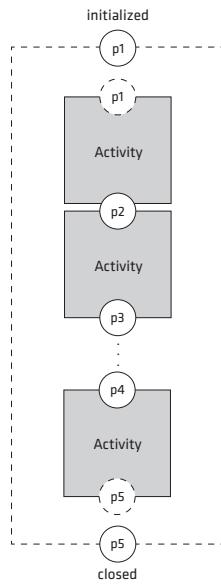


Abbildung 11: Pattern der **Sequence**-Aktivität

Parallel

Die **Parallel**-Aktivität plant im Gegensatz zur sequentiellen Durchführung der **Sequence**-Aktivität eine nebenläufige Abwicklung von verschiedenen Kindaktivitäten. Die Aktivität ist beendet, wenn alle enthaltenen Aktivitäten ihre Ausführung beendet haben. Eine optionale **CompletionCondition** erlaubt es eine Bedingung zu definieren, die jeweils nach Ablauf einer Kindaktivität geprüft wird und bei Zutreffen alle noch laufenden Aktivitäten stoppt (vgl. [Buk10, S.105]). Das Pattern in Abb.12 bildet wegen den zuvor definierten Einschränkungen diese Abbruchbedingung noch nicht ab, sondern lediglich die normale Beendigung aller Aktivitäten. In der tatsächlichen Umsetzung in WF handelt es sich jedoch nicht um eine echt parallele Ausführung, da jede Workflow-Instanz nur in einem Thread läuft und somit nur eine Pseudo-Parallelität erreicht werden kann. Zu Beginn der Aktivität werden alle Zweige zur Ausführung vorbereitet und anschließend mit dem ersten begonnen. Gibt dieser jedoch nie die Ausführung ab, d.h. er wird inaktiv, kann der folgende Zweig erst nach Beendigung des ersten ausgeführt werden, was einer normalen **Sequence** entspräche. Inaktiv werden kann eine Aktivität immer dann, wenn sie auf eine externe Eingabe wartet. Dies kann z.B. durch einen Nachrichtenempfang oder auch durch eine **Delay**-Aktivität geschehen. Erst dann kann sich der Ablauf zwischen den Zweigen ändern (vgl. [Buk10, S.176 ff.]).

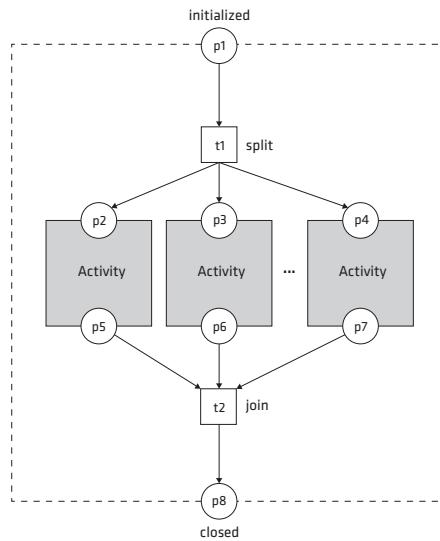


Abbildung 12: Pattern der Parallel-Aktivität

If

Die If-Aktivität kann benutzt werden, um eine bedingte Verzweigung innerhalb eines Workflows zu modellieren. Hierzu wird ein boolescher Ausdruck definiert, aufgrund dessen Wahrheitswertes einer der Zweige ausgeführt wird. Ist der Ausdruck wahr, so wird der *Then*-Fall beschritten, andernfalls der *Else*-Zweig (vgl. [Buk10, S.164]). Abbildung 13 zeigt das Pattern. Im Petri-Netz wird die Verzweigung aufgrund der gleichartigen Token nichtdeterministisch entschieden.

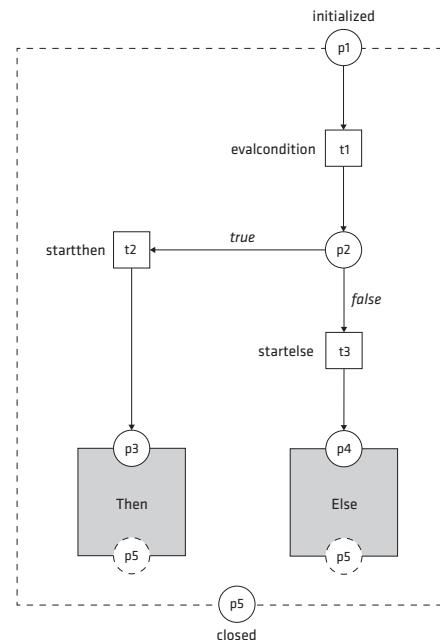


Abbildung 13: Pattern der If-Aktivität

Switch

Im Gegensatz zur auf einem booleschen Ausdruck basierenden **If**-Aktivität ist es mit der **Switch<T>**-Aktivität möglich, Entscheidungen aufgrund von verschiedenen Variablenausprägungen zu treffen. Dies geschieht durch Definition von unterschiedlichen Fällen. Da es eine generische Aktivität (vom Typ T) ist, muss der Typ der auszuwertenden Variable festgelegt werden. Alle möglichen Fälle müssen von eben diesem Typ sein. Weiterhin kann ein zusätzlicher *Default*-Fall definiert werden, der ausgeführt wird falls keiner der definierten Fälle zutrifft. Die Fälle sind dabei alle eindeutig, d.h. es können nicht zwei Fälle gleichzeitig zutreffen (vgl. [Buk10, S.164]). Das Pattern in Abb.14 zeigt die Umsetzung für zwei Fälle. Alle Fälle werden der Definitionsreihenfolge nach abgearbeitet. Trifft der Ausdruck zu, wird die entsprechende Fallaktivität gestartet und anschließend die **Switch<T>**-Aktivität beendet, ansonsten wird der nächste Ausdruck ausgewertet. Wenn sich keiner der Fälle bewahrheitet, wird schlussendlich die Default-Aktivität aufgerufen.

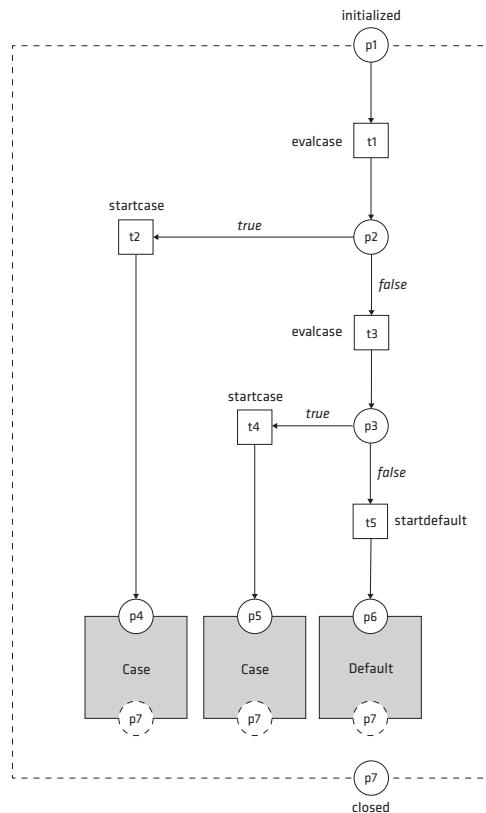


Abbildung 14: Pattern der **Switch**-Aktivität

Pick

Eine **Pick**-Aktivität beinhaltet beliebig viele Zweige (sog. *PickBranches*), deren Aktivitäten basierend auf dem Eintreten eines Ereignisses, dem sog. *Trigger*, ausgeführt werden. Jeder PickBranch besteht dabei aus einer Trigger-Aktivität und einer Action-Aktivität. Letztere wird ausgeführt, wenn das zugehörige Triggerereignis vollständig abgeschlossen wurde (vgl. [Buk10, S.105]). Im Vergleich zur äquivalenten BPEL-Aktivität, die lediglich den Empfang einer Nachricht sowie eine definierte Zeitdauer oder einen Zeitpunkt als

Trigger erlaubt (vgl. [OAS07, S.100 f.]), kann in WF jede mögliche Aktivität als Trigger verwendet werden. Konzeptionell gesehen werden alle Trigger der vorhandenen PickBranches parallel ausgeführt. Der PickBranch, dessen Trigger zuerst vollständig beendet, wird auch tatsächlich ausgeführt. Alle anderen Trigger, die möglicherweise bereits einen Teil ihrer Logik ausgeführt haben, werden abgebrochen. Dieses Konzept führt potentiell zu Problemen, beispielsweise in einer **Pick**-Aktivität mit zwei Triggern, in der jeder Trigger aus einer **Receive**-Aktivität gefolgt von weiteren Aktivitäten besteht. Wenn die zusätzliche Logik den Workflow inaktiv werden lässt (z.B. durch eine **Delay**-Aktivität), besteht die Möglichkeit, dass beide **Receive**-Aktivitäten unterschiedlicher Trigger erfolgreich ausgeführt werden. Einer der Trigger wird anschließend vollständig beendet und die Ausführung des anderen abgebrochen. In den meisten Fällen wird der Empfang der Nachricht und eine teilweise Ausführung der Triggerlogik nicht akzeptabel sein. Vor allem, da zudem nicht klar ist, welche Teile der Aktivitäten ausgeführt wurden. Es wird deshalb empfohlen, dass der Trigger so wenig Logik wie möglich enthält. Alle weiteren Aktivitäten sollten in der Action des PickBranches untergebracht werden (vgl. [Mic12k], [Buk10, S.302]). Da die Petri-Netz-Pattern keine Abbruchbedingungen implementieren, wird das Netz so modelliert als könnte immer nur ein Trigger feuern. Dies deckt sich auch mit den vorab genannten Richtlinien zur Nutzung der Aktivität. Durch dieses Verhalten wird zudem auch nicht steuerbares Verhalten durch partielles, für Partner nicht beobachtbares Ausführen von Triggern innerhalb der Netze verhindert. Das Pattern in Abb.15 zeigt die aktuelle Umsetzung anhand zweier PickBranches. Beide initialen Stellen der Trigger sind mit der initialen Stelle der Aktivität verschmolzen, was dazu führt, dass nur eine der Trigger-Aktivitäten ausgeführt werden kann. Nach Beendigung eines Triggers wird direkt die zugehörige Action gestartet.

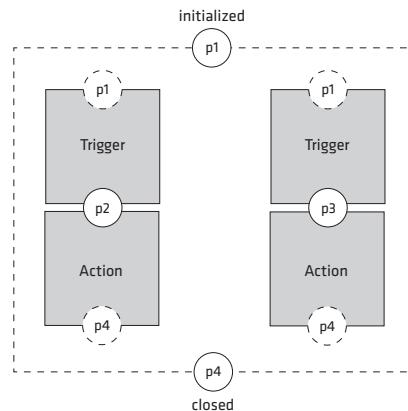


Abbildung 15: Pattern der **Pick**-Aktivität

While und DoWhile

Ein weiteres geläufiges Workflow-Konstrukt sind Schleifen. Um Schleifen in sequentiellen Workflows zu verwenden, stehen die beiden Aktivitäten **While** und **DoWhile** zur Verfügung. Die **While**-Aktivität führt eine Kindaktivität solange kontinuierlich aus, wie ein boolescher Ausdruck wahr ist. Nachdem der Ausdruck bereits zu Beginn falsch sein könnte, kann es sein, dass die Aktivität nie ausgeführt wird. Im Gegensatz dazu wertet die **DoWhile**-Aktivität den Ausdruck erst aus, nachdem die Aktivität ausgeführt wurde. Dadurch wird die Aktivität zumindest einmal ausgeführt. Die Ausführung stoppt ebenso wenn der Ausdruck falsch ist (vgl. [Buk10, S.104, 165]). Abbildung 16 zeigt die beiden

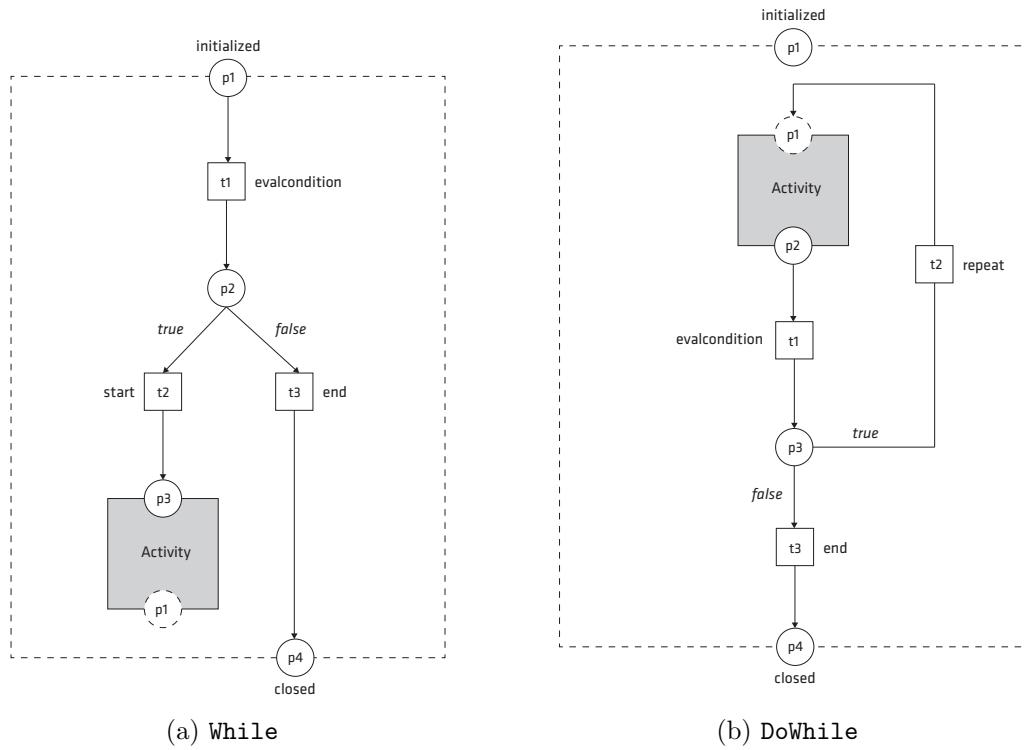


Abbildung 16: Patterns der Schleifen–Aktivitäten

Patterns. Bei **While** (siehe Abb.16a) wird zunächst der Ausdruck ausgewertet und dann entweder die Schleife beendet oder die Aktivität ausgeführt. Nach Ausführung der Aktivität wird der Kontrollfluss zurück in die initiale Position gebracht und der Ablauf wiederholt. Bei **DoWhile** (siehe Abb.16b) wird im Gegensatz dazu zunächst die Aktivität ausgeführt und dann der Ausdruck überprüft. Je nach Wert wird der Kontrollfluss wieder zurück in die initiale Position gebracht oder die Aktivität beendet.

4.2 Flowchart Workflows

Neben der sequentiellen Modellierung unterstützt WF auch eine graphbasierte Modellierung von Workflows durch sog. **Flowcharts**. Das Besondere an dieser Technik ist, dass direkte Verbindungen zwischen Aktivitäten definiert werden, um den Fluss der Ausführung zu bestimmen. Dies steht im Gegensatz zu den Kontrollfluss-Aktivitäten (z.B. **If**, **While**), die bei der sequentiellen Modellierung benutzt werden, um Verzweigungen und Schleifen zu erzeugen. **Flowcharts** sind in WF Aktivitäten, genauso wie die bereits vorgestellten sequentiellen Konstrukte. Sie können deshalb in jedem Modellierstil den Platz einer Aktivität einnehmen, was eine Mischung der verschiedenen Stile erlaubt. Ein **Flowchart** besteht aus beliebig vielen verschiedenen **FlowNodes**, die miteinander verbunden sind. Ein **FlowNode** stellt hierbei eine abstrakte Oberklasse für die konkreten Ausprägungen **FlowStep**, **FlowDecision** und **FlowSwitch** dar. **FlowStep** bezeichnet einen Container für die Schritte in einem **Flowchart**, der eine beliebige auszuführende Aktivität und eine optionale Verbindung **FlowStep.Next** zu einem anderen **FlowNode** beinhalten kann. Für den Entwickler ist dies jedoch transparent, da immer direkt Aktivitäten im Designer miteinander verbunden werden. Die beiden Aktivitäten **FlowDecision** und **FlowSwitch** sind Konstrukte, um konditionale Anweisungen im Stile von **If** (siehe 4.1.2), respektive **Switch<T>** (siehe 4.1.2), im Kontext eines **Flowcharts** zu nutzen. Der obligatorische **StartNode** identifiziert den Einstiegsknoten des Graphen (vgl. [Buk10, S.229 f.]). Abbildung 17 zeigt ein beispielhaftes Petri-Netz-Pattern eines **Flowcharts** in der Übersicht.

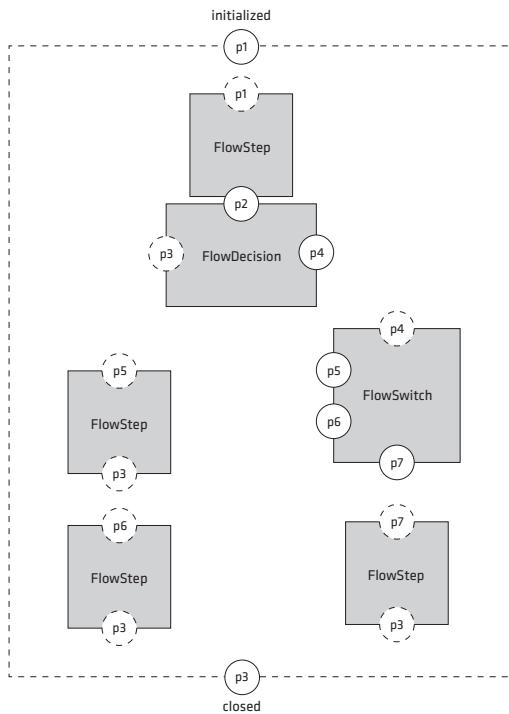


Abbildung 17: Beispiel-Pattern eines **Flowcharts**

Der Kontrollfluss innerhalb des **Flowcharts** wird durch Verschmelzung der enthaltenen **FlowNodes** abgebildet. Ein **FlowStep** beinhaltet dabei lediglich die im Teilschritt auszuführende Aktivität (siehe Abb.18). Als Aktivität eines **FlowSteps** können beliebige sequentielle Aktivitäten, eine **StateMachine**, oder wiederum ein **Flowchart** zum Einsatz kommen. Die Kontrollflusskonstrukte **FlowDecision** und **FlowSwitch** sind keine eigenen

Aktivitäten und können auch nur innerhalb eines **Flowcharts** benutzt werden. Sie besitzen deshalb auch nicht das allgemein gültige Petri–Netz–Interface. Ist ein **FlowNode** mit keinem weiteren Knoten verbunden, so wird der gesamte **Flowchart** nach der Ausführung des Knotens beendet. Die inneren Teilnetze sowie eine genauere Beschreibung der anderen **FlowNodes** werden in den beiden folgenden Unterkapiteln beschrieben.

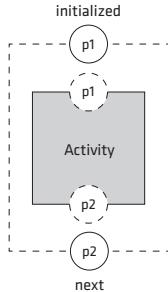


Abbildung 18: **FlowStep**–Pattern

4.2.1 FlowDecision

Die **FlowDecision** ermöglicht es, den Kontrollfluss in einem Flowchart durch einen einfachen booleschen Ausdruck zu verzweigen. Je nach Wahrheitswert des Ausdrucks können zwei verschiedene Aktivitäten (jeweils in einem **FlowStep** eingebettet) ausgeführt werden. Beide zur Verfügung stehenden Zweige sind optional, d.h. im Falle, dass für einen Wahrheitswert keine Aktivität definiert wurde, endet der **Flowchart** und es werden keine weiteren Aktivitäten ausgeführt (vgl. [Buk10, S.231]). Das Pattern in Abbildung 19 beinhaltet eine einfache Fallunterscheidung, die zu den beiden Ausgangsstellen führt, welche mit den gewünschten **FlowNodes** verschmolzen werden können.

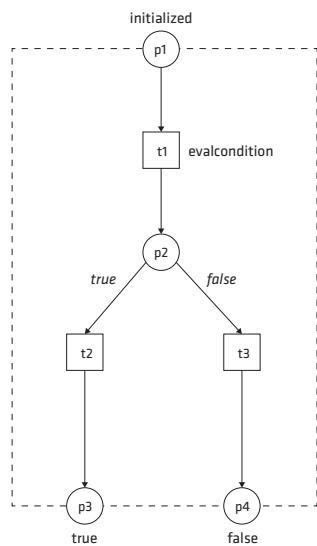


Abbildung 19: **FlowDecision**–Pattern

4.2.2 FlowSwitch<T>

Das **FlowSwitch<T>**-Konstrukt funktioniert genauso wie die **Switch<T>**-Aktivität (siehe 4.1.2). Es erlaubt einen Ausdruck zu definieren, aufgrund dessen möglicher Werte verschiedene Zweige ausgeführt werden. Da **FlowSwitch<T>** ebenso wie **Switch<T>** generisch ist, muss der Typ des Ausdrucks vorher definiert werden und jeder Wert, der eine Verzweigung definiert muss typgleich sein. Für jeden definierten Fallwert existiert eine Ausgangsstelle, die mit einem **FlowNode** verbunden werden kann. Weiterhin kann wiederum auch für den Default-Fall ein **FlowNode** spezifiziert werden, ansonsten endet die Ausführung des **Flowcharts** (vgl. [Buk10, S.231]). Abbildung 20 zeigt das entsprechende Pattern. Äquivalent zur **Switch<T>**-Aktivität werden alle Fälle nacheinander bearbeitet und schlussendlich der Default-Fall ausgeführt.

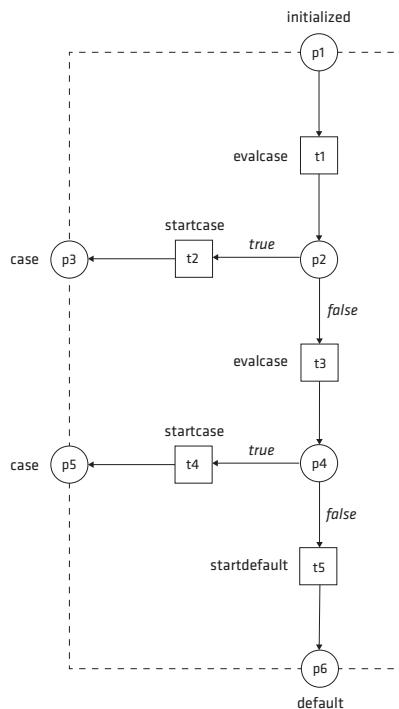


Abbildung 20: **FlowSwitch**-Pattern

4.3 StateMachine Workflows

Mit dem Platform Update 1¹⁴ wurde WF 4 die Möglichkeit einer **StateMachine**–Modellierung hinzugefügt. Endliche Automaten unterscheiden sich in vielen Punkten von den beiden vorausgegangenen Modellierungstypen. Sequentielle Workflows definieren eine fixierte Abfolge von auszuführenden Schritten. Ein endlicher Automat hingegen definiert eine Menge von Anwendungszuständen und mögliche Übergänge zwischen diesen. Eine Instanz eines Workflows kann sich dabei immer genau in einem Zustand befinden. Jeder Zustand kann auf bestimmte Ereignisse reagieren, die einen Übergang zu einem anderen Zustand auslösen können. Die Zustände können ferner Aktivitäten beinhalten, die vor einem Übergang in einen neuen Zustand ausgeführt werden. Wegen ihres dynamischen Kontrollflusses eignen sich endliche Automaten hervorragend in Situationen, die Aktionen anhand von externen Ereignissen, wie z.B. auch schwer voraussagbaren menschlichen Interaktionen, durchführen (vgl. [Buk08, S.333 f.]). Ähnlich wie bei **Flowcharts** wird der Kontrollfluss bei **StateMachines** graphbasiert modelliert. Die Konzepte der beiden Modellierungsweisen unterscheiden sich jedoch stark. Während im endlichen Automaten ein Zustand einen genauen Status eines Workflows beschreibt, wird dieser im **Flowchart** in einem bestimmten **FlowStep** meist nur mit Kenntnis der vorausgegangenen Schritte klar. Ausschlaggebend für den Ablauf des Kontrollflusses sind bei endlichen Automaten im Allgemeinen Ereignisse, während der **Flowchart** internen Entscheidungen (z.B. **FlowDecision**) und der definierten Sequenz folgt. Zudem wird die konkrete Arbeit innerhalb eines **FlowSteps** durchgeführt, während ein endlicher Automat innerhalb eines Zustands inaktiv ist und auf ein Ereignis wartet, das die Verarbeitung der Logik im Zuge eines Übergangs verrichtet (vgl. [Buk10, S.229 ff.], [Mic12i]). Genauso wie ein **Flowchart** ist eine **StateMachine** in WF eine Aktivität, die in jedem Modellierungsstil an entsprechender Stelle verwendet werden kann. Abb.21 zeigt die höchste Abstraktionsstufe eines Petri-Netz–Patterns einer möglichen **StateMachine**. Sie enthält vier Zustände (**States**), die untereinander verbunden sind. Die Verbindungen stellen mögliche Übergänge zwischen den Zuständen dar.

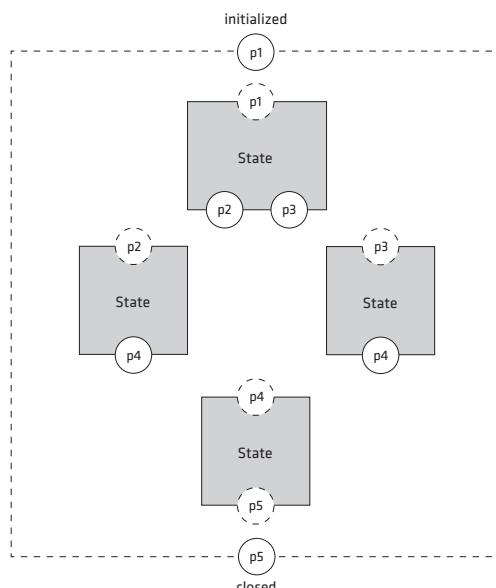


Abbildung 21: Beispiel–Pattern einer **StateMachine**

¹⁴Siehe <http://msdn.microsoft.com/en-us/library/hh290669>

4.3.1 State

Abbildung 22 geht eine Abstraktionebene tiefer als Abb.21. Bei Eintritt in einen **State** wird zunächst eine Entry–Aktivität ausgeführt. Anschließend befindet sich der Automat im Wartezustand. Tritt ein Triggerereignis der definierten Transitionen (**Transition**) ein, so wird die jeweilige Transition ausgeführt. Die Interna der Transition werden im folgenden Unterkapitel beschrieben, da mehrere Varianten existieren. Genauso wie in Kapitel 4.1.2 wird hier die Annahme getroffen, dass sinnvollerweise nur ein Trigger feuern kann. Die dort ausgeführten Begründungen gelten analog für diesen Fall. Nachdem die Transition erfolgreich beendet wurde, wird die Exit–Aktivität des Zustandes aufgerufen, die bei jedem Verlassen eines Zustandes ausgeführt wird. Gleichzeitig wird eine Wartestelle belegt, die auf Beendigung der Exit–Aktivität wartet und das Netz anschließend in den neuen Zustand überführt. Handelt es sich bei einem **State** um einen finalen Zustand, so enthält dieser lediglich die Entry–Aktivität und terminiert anschließend. Folglich ist der Zustand zusätzlich mit der *closed*–Interfacestelle der **StateMachine** verbunden. Auf eine explizite Darstellung wird verzichtet (vgl. [Mic12i, Jac12]).

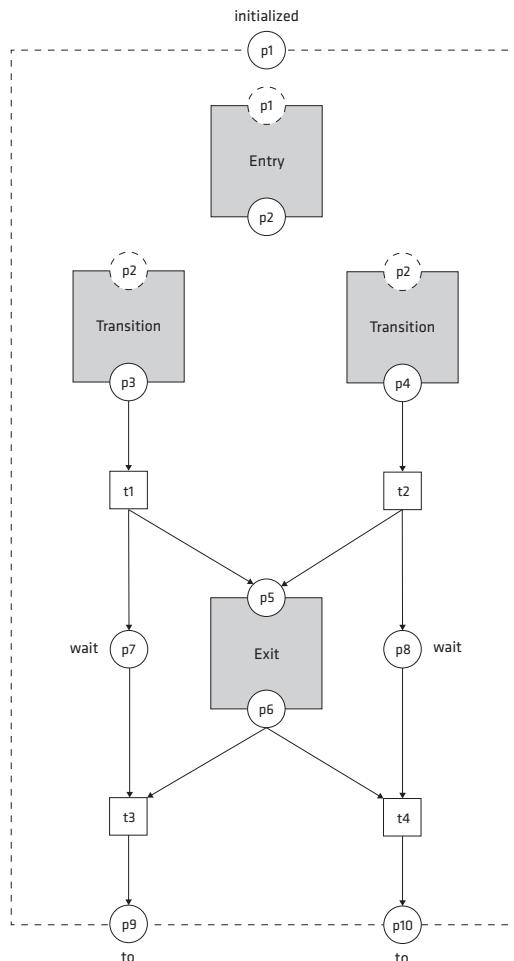


Abbildung 22: State–Pattern

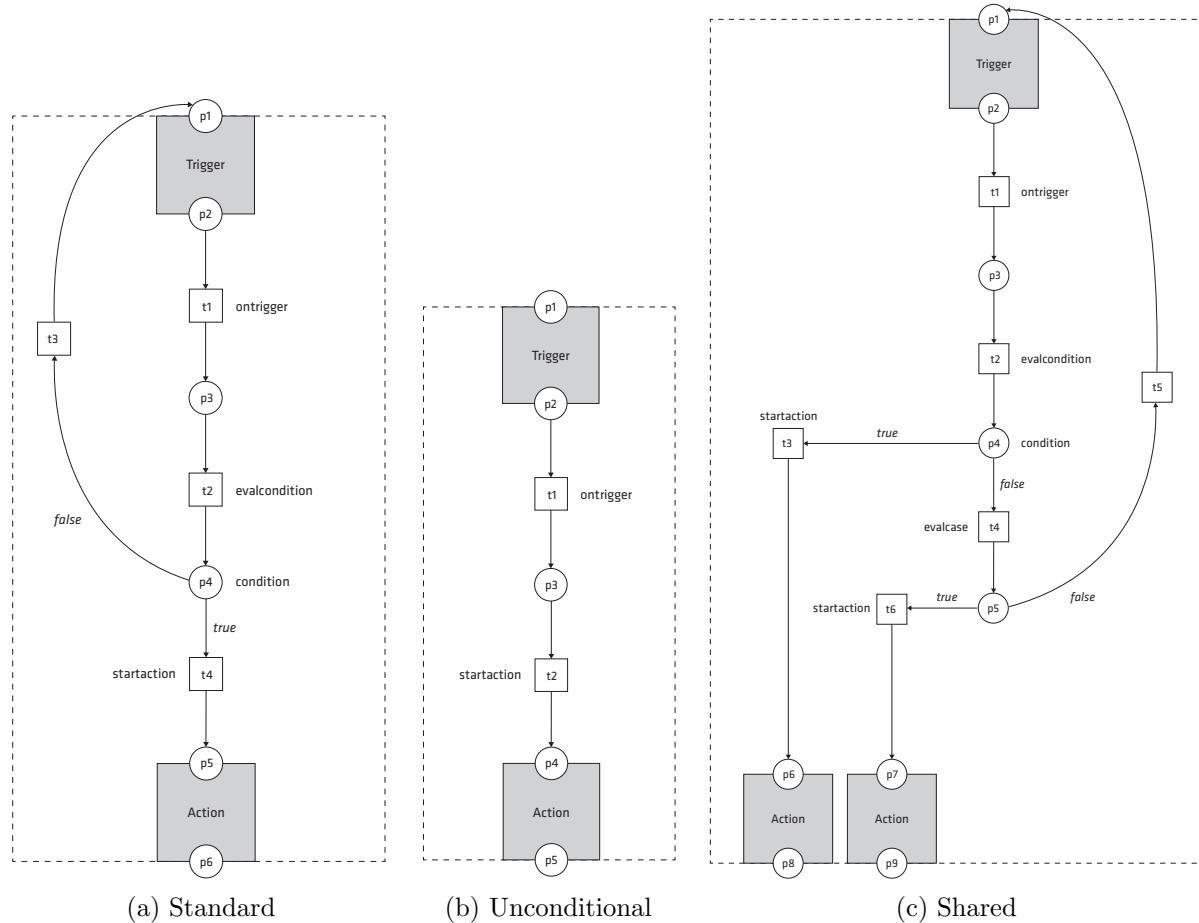


Abbildung 23: Transition-Patterns

4.3.2 Transition

Eine **Transition** besteht standardmäßig aus einem **Trigger**, der die auslösende Ereignis-Aktivität enthält, einem Ausdruck, der geprüft wird bevor der Übergang durchgeführt wird und einer Action-Aktivität, die im Erfolgsfall ausgeführt wird. Es existieren einige wichtige Abwandlungen des Standardfalls, die zusammen mit diesem in Abb.23 dargestellt sind.

Abb.23a zeigt den angesprochenen Standardfall. Die **Transition** wird ähnlich der **Pick**-Aktivität durch eine **Trigger**-Aktivität angestoßen. Zusätzlich wird anschließend der definierte Ausdruck ausgewertet. Trifft dieser zu, wird die **Action** der **Transition** ausgeführt und anschließend im zugehörigen **State** die **Exit**-Aktivität. Trifft er nicht zu, wird der Kontrollfluss zurück zur initialen Stelle geführt und auf ein erneutes Triggerereignis gewartet. Es ist auch möglich, dass sich mehrere **Transitions** eine **Triggeraktivität** teilen. Abbildung 23c zeigt diesen sog. *Shared Trigger*. Die Ausführung der **Transition** wird anhand des Ausdrucks entschieden. Die Verzweigung ist äquivalent zum Pattern der **Switch<T>**-Anweisung. Alle Fälle werden dabei nach der Definitionsreihenfolge ausgewertet und ggf. der Übergang gestartet. Trifft kein Fall zu, wird der Kontrollfluss wiederum zurückgeführt und ein erneutes Ereignis erwartet. Existiert keine Bedingung zur Prüfung des Übergangs, so handelt es sich um einen sog. *Unconditional Trigger* (siehe Abb.23b). Dieser führt nach Beendigung des Triggers sofort seine Aktivität aus und leitet den Über-

gang in den nächsten Zustand ein. Zusätzlich ist es möglich, in jeder der drei Varianten auf die Triggeraktivität zu verzichten. Dieser *Empty* oder *Null Trigger* wird sofort nach der Entry–Aktivität des Zustands ausgeführt. Verwendet man einen Null Trigger in Kombination mit einem Unconditional Trigger, so wird diese Transition zum nächsten Zustand bedingungslos nach Ausführung der Zustandsaktivität abgewickelt. Um Konflikte zu vermeiden, darf jeder Zustand nur eine solche Transition besitzen (vgl. [Mic12i, Jac12]).

4.4 Einschränkungen

Im Folgenden werden die nicht betrachteten Aktivitäten aufgeführt. Wenn spezielle Gründe für eine Nichtbeachtung existieren, werden diese knapp dargelegt. Andernfalls erfolgt der Vollständigkeit halber lediglich eine kurze Nennung der Aktivität.

Die `ForEach<T>`–Aktivität führt sequentiell eine definierte Aktivität für jedes Element einer Auflistung des Typs T aus. Die `ParallelForEach<T>` verfährt genauso, jedoch werden alle Aktivitäten parallel abgewickelt. Die Schleifendurchläufe, bzw. die Anzahl der Zweige in der parallelen Bearbeitung, sind von der Anzahl der in der Auflistung enthaltenen Elementen abhängig. Eine statische Analyse wird in den meisten Fällen nicht genügen, um die Anzahl der Elemente zu ermitteln, da diese meist erst zur Laufzeit feststeht. Dieses nicht triviale datenbasierte Problem ist mit hoher Wahrscheinlichkeit nicht entscheidbar und benötigt eine gesonderte Betrachtung. WF stellt zur deklarativen Nutzung von Auflistungen in Workflows einige vorgefertigte Aktivitäten zur Verfügung. Da keinerlei Datenaspekte von Workflows betrachtet werden, sind auch diese primitiven Hilfsaktivitäten nicht implementiert. Ein entsprechendes Pattern wäre unter den aktuellen Voraussetzungen äquivalent zu den gezeigten Primitiven. Es handelt sich dabei um die Aktivitäten `AddToCollection<T>`, `ClearCollection<T>`, `ExistsInCollection<T>` und `RemoveFromCollection<T>`. Im Zuge der Datenabstraktion wird auch die Korrelation nicht mit modelliert. Dies betrifft die Aktivitäten `CorrelationScope` und `InitializeCorrelation`. Da aktuell nur der fehlerfreie Ablauf der Aktivitäten behandelt wird, werden die Aktivitäten `Rethrow`, `Throw` und `TryCatch` nicht betrachtet. Dies schließt auch Abbrüche und anschließende Compensation–Aktivitäten wie `CancellationScope`, `CompensableActivity`, `Compensate`, `Confirm` und `TerminateWorkflow` mit ein. Ebenso werden keine Transaktionen (`TransactedReceiveScope`, `TransactionScope`) behandelt. Die primitive Aktivität `InvokeMethod` ermöglicht es, eine öffentliche Methode einer Klasse oder Objektinstanz aufzurufen. Da die Implikationen des Aufrufs bzw. das Petri–Netz, welches das Verhalten der Methode abbildet, nicht statisch zu ermitteln sind, muss auf eine Unterstützung verzichtet werden. Die `Persist`–Aktivität kann dazu benutzt werden, den Zustand einer Workflow–Instanz explizit zu speichern. Der Zustand des Petri–Netzes bleibt davon unberührt. Um Aktivitäten, die mit WF Version 3.x erstellt worden sind, weiterzuverwenden, steht die `Interop`–Aktivität zur Verfügung (vgl. [Buk10, S.104-110]).

5 Compiler Prototyp WF2oWFN

Die in Kapitel 4 gezeigten Pattern sind im Prototypen *WF2oWFN*¹⁵ implementiert. Der Compiler ermöglicht es, aus WF-Prozessen kontrollflusskonforme Petri-Netze zu generieren. Er ist der erste Schritt zur automatisierten Transformation von WF-Prozessen in Petri-Netze und zur Eingliederung in die bestehende Werkzeugkette zur Analyse von Verhaltensaspekten von Services mit Hilfe von Petri-Netzen¹⁶. Gleichzeitig stellt er den Proof of Concept der vorgestellten Theorien dar. Um die Korrektheit und Funktionsfähigkeit des Compilers zu überprüfen wurden insgesamt 137 Tests geschrieben, ausgeführt und manuell verifiziert¹⁷. Darunter befinden sich auch sämtliche WF-Prozesse, die aus einer Teilmenge der in Kapitel 4 aufgeföhrten Aktivitäten bestehen, welche im Zuge einer patternbasierter Analyse von WF [Len11] erstellt wurden. In der Folge wird die Implementierung des Compilers genauer vorgestellt.

5.1 Architektur

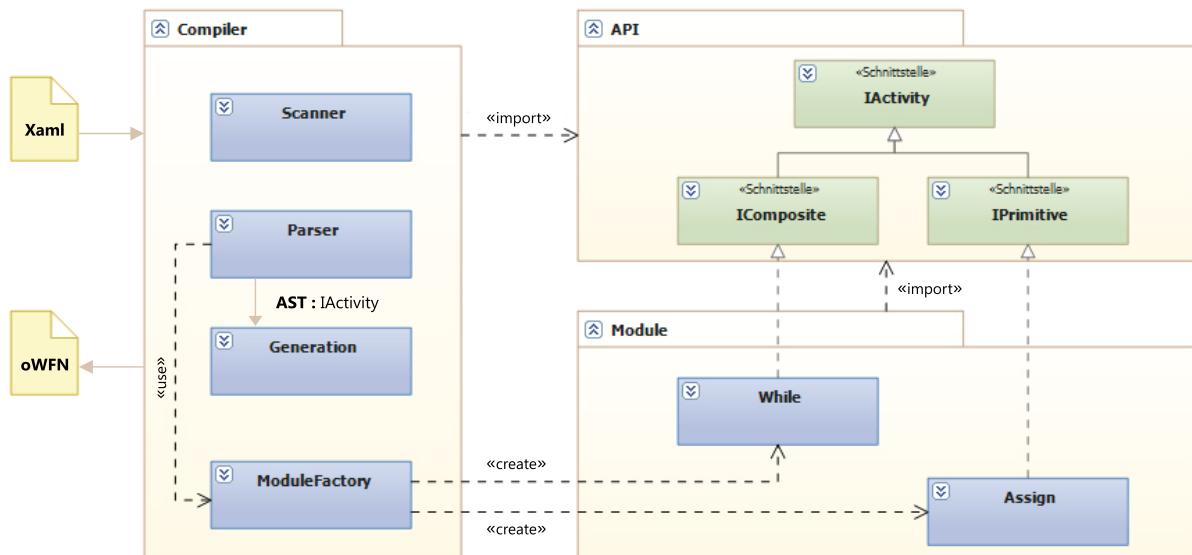


Abbildung 24: Grobübersicht der Funktionsweise des Compilers

Abbildung 24 zeigt das Konzept des Compilers zunächst in einer vereinfachten Übersicht. Als Input können beliebige WF-Prozesse der Version 4.x verwendet werden, die ausschließlich aus den in Kapitel 4 vorgestellten Aktivitäten bestehen. Dabei werden sowohl Workflows (*.xaml) als auch Workflow Services (*.xamlx) unterstützt. Das Paket *Compiler* beinhaltet übliche Schritte eines Compilers (siehe [ALSU07, S.4–12]). Darunter der *Scanner*, der die lexikalische Analyse der Xaml(x)-Dateien übernimmt. Dabei zerlegt dieser den eingelesenen Quelltext in eine Folge von Token, die einzelne Aktivitäten und ihre Eigenschaften darstellen. Diese Token werden vom *Parser* in einen abstrakten

¹⁵Der Quellcode sowie eine kompilierte Version des Prototyps befindet sich auf dem beiliegenden Datenträger im Ordner WF2OWFN.

¹⁶Publikationen und Tools erhältlich unter <http://service-technology.org>

¹⁷Die Abbildungen der resultierenden Petri-Netze sowie alle gescreenten Xaml(x)-Dateien befinden sich auf dem Datenträger im Ordner TESTS.

Syntaxbaum umgewandelt. Jeder Knoten des Baumes stellt dabei eine Aktivität des WF–Prozesses dar. Falls während dieser syntaktischen Analyse unerwartete Token auftreten, gibt der Parser einen Syntaxfehler aus. Da in WF die Menge der verfügbaren Aktivitäten nicht limitiert ist, stellt sich die Herausforderung einer einfachen Erweiterbarkeit des Compilers um neue Aktivitäten. Solche *CustomActivities* können in WF jederzeit durch Code oder durch deklarative Orchestrierung vorhandener Aktivitäten geschaffen werden (vgl. [Buk10, S.46 f.]). Aus diesem Grund ist der Compiler so geschaffen, dass er ein *Application Programming Interface (API)* zur Verfügung stellt, mit dem Module für neue Aktivitäten durch einen Pluginmechanismus hinzugefügt werden können. Für jede Aktivität existiert eine Klasse, die sowohl die Fähigkeit des Parsens einer Aktivität als auch die Generierung der entsprechenden Petri–Netz–Repräsentation übernimmt. Alle Module müssen dabei von dem Interface **IActivity** des APIs erben. Je nachdem, ob es sich um eine primitive Aktivität oder um eine strukturierte Aktivität, die weitere innere Aktivitäten enthält, handelt, ist das Modul vom konkreten Typ **IPrimitive** bzw. **IComposite**. Die Module werden als Klassenbibliotheken kompiliert und zur Laufzeit vom Compiler geladen. Hierfür existiert die **ModuleFactory**, die alle verfügbaren Assemblies verwaltet und als **IActivity** instanzieren kann. Der Parser kann die Factory verwenden, um ein Modul für eine entsprechende Aktivität zu instanzieren. Die Factory wird dabei auch an alle von **IComposite** abgeleiteten Module per *Dependency Injection* weitergegeben. Diese erlangen dadurch die Möglichkeit, selbst eine Instanz eines entsprechenden Moduls für ihre inneren Aktivitäten aufzurufen. Als Ergebnis des Rücklaufes entsteht der Syntaxbaum des Prozesses. Innerhalb der *Generation* wird nun für jeden Knoten des Syntaxbaumes die Petri–Netz–Generierung ausgeführt und die entstandenen Netze jeweils in der übergeordneten Instanz eines Moduls zusammengefügt. Die anschließende Ausgabe des Petri–Netzes erfolgt standardmäßig als oWFN. Ferner werden weitere Ausgabeformate unterstützt. Die einzelnen Komponenten des skizzierten Prozesses werden nun im Detail betrachtet.

5.1.1 API

Das API dient dazu, unabhängig vom Quellcode des Compilers Module schreiben zu können, die WF–Aktivitäten parsen und in eine Petri–Netz–Repräsentation überführen. Der Compiler ist hierdurch einfach um zusätzliche Aktivitäten erweiterbar. Zu diesem Zweck besteht das API aus einer Sammlung von Klassen, um Petri–Netz–Strukturen zu erstellen und zu manipulieren, die sich im Namespace *WF2oWFN.API.Petri* befinden. Weiterhin sind verschiedene Interfaces für die Module innerhalb des Basisnamespaces *WF2oWFN.API* enthalten. Abbildung 25 zeigt das API im Überblick.

WF2oWFN.API.Petri

Die Basis für die Petri–Netze bilden die Klassen des Pakets *WF2oWFN.API.Petri*. Es enthält alle nötigen Funktionen, um Stellen/Transitions–Netze zu erstellen und zu manipulieren. Die abstrakte Oberklasse **Node** für die Knoten eines Netzes enthält die gemeinsamen Attribute der konkreten Klassen für Stellen und Transitionen. Sämtliche Attribute sind als *Properties*¹⁸ implementiert. Darunter ein eindeutiger *Name* des Knotens, inklusive eines Verlaufs vorheriger Rollen (*History*), die durch Stellenverschmelzungen entstanden

¹⁸Properties sind eine C#–spezifische Syntax ähnlich zu konventionellen Getter/Setter–Methoden. Eigenschaften können dabei wie öffentliche Attribute verwendet werden (siehe [Mic12l]).

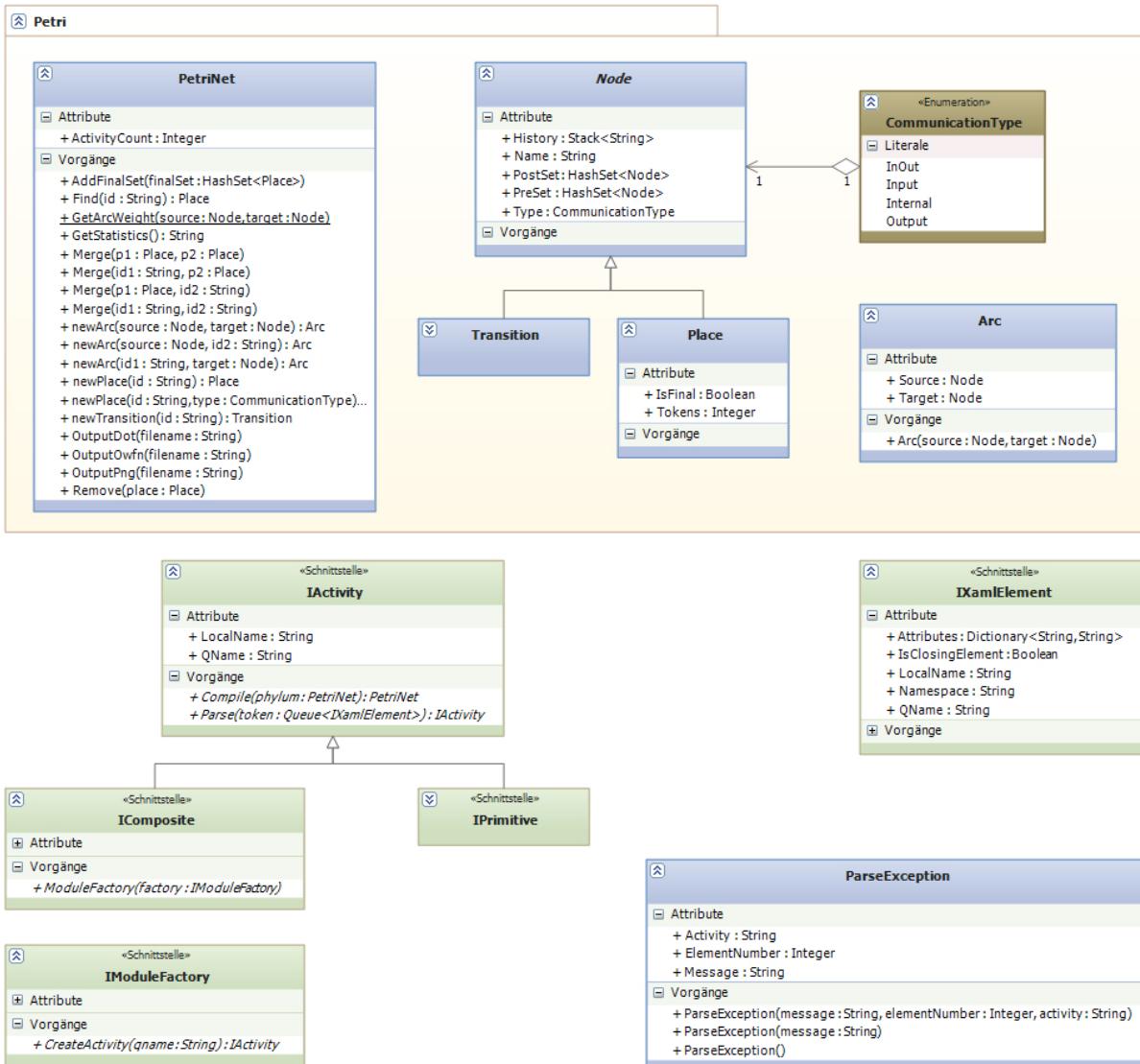


Abbildung 25: Das API von WF2oWFN

sind. Zudem werden alle Vorgänger- und Nachfolgerknoten sowie der Typ des Knotens gespeichert. Der Kontrollfluss innerhalb des Netzes wird durch Verbindungen (Klasse **Arc**) zwischen den Knoten abgebildet. Die Klasse **PetriNet** verwendet die Strukturen und organisiert das gesamte Petri–Netz. Dies schließt die wichtigsten Funktionen der Erstellung, Löschung und der Suche nach Stellen und Transitionen ein. Weiterhin besteht die Möglichkeit Stellen zu verschmelzen und das Netz in diversen Ausgabeformaten zu persistieren. Standardmäßig erfolgt die Ausgabe der Petri–Netze im oWFN–Format¹⁹ und eine Ausgabe als *Portable Network Graphic (PNG)* unterstützt. Letztere werden über die Nutzung des Tools PNAPI²⁰ und Graphviz ermöglicht, während die oWFN–Ausgabe nativ anhand des in [Loh10, S.10] gegebenen BNF–Schemas implementiert ist. Der Funktionsumfang der Petri–Netz–Klassen beschränkt sich aktuell auf die im Compiler benötigten Funktionen. Das Paket stellt im Gegensatz zur PNAPI noch kein eigenständig nutzbares, abgeschlossenes API dar.

¹⁹Graphviz ist eine Open–Source–Software zur Visualisierung von Graphen. Dokumentation und Download unter www.graphviz.org

²⁰Dokumentation und Download unter <http://download.gna.org/service-tech/pnapi/>

WF2oWFN.API

Der Namespace `WF2oWFN.API` beinhaltet die Interfaces der Module sowie weitere abhängige Schnittstellen. Das Basisinterface aller Module ist das Interface `IActivity`. Es repräsentiert die Abbildung einer WF–Aktivität. `IActivity` stellt Methoden zum Parsen einer Xaml–Aktivität und zum Kompilieren einer Petri–Netz–Repräsentation bereit. Jede Aktivität und damit auch das entsprechende Modul wird über seinen qualifizierten Namen (Attribut `QName`) referenziert. Der `QName` repräsentiert dabei eine eindeutige Identität der Aktivität, bestehend aus deren Namespace und dem lokalen Elementnamen (in der Form `{Namespace}LocalName`). Je nachdem, ob es sich bei der Aktivität um eine strukturierte oder eine primitive Aktivität handelt, ist sie vom konkreten Typ `IComposite` oder `IPrimitive`. `IPrimitive` fungiert hierbei lediglich als Marker–Interface und stellt keine weiteren Methoden bereit. `IComposite` hingegen fordert eine weitere Setter–Methode, innerhalb einer Property, für die Übergabe einer `IModulFactory` zur Behandlung innerer Aktivitäten. Eine Instanz einer `IModulFactory` wird vom Compiler initial mit allen im Unterordner `MODULES` befindlichen Modulen befüllt. Um neue Aktivitäten hinzuzufügen genügt es, das kompilierte Modul in den Ordner zu kopieren. Der Methode `Parse` wird jeweils eine Queue mit den zu analysierenden Xaml–Tokenen übergeben. Der Typ der Token wird durch das Interface `IXamlElement` beschrieben. Es enthält den `QName`, die Attribute des Elements und die Information ob es sich um ein schließendes Element handelt. Zur Weitergabe von Exceptions während des Parsens steht eine benutzerdefinierte Exception (`ParseException`) zur Verfügung. Zum Zwecke eines besseren Debuggings enthält sie, neben der standardmäßigen Nachricht, Attribute für den Namen der bearbeiteten Aktivität und die Elementnummer des auslösenden Tokens in der verarbeiteten Queue. Nach Bearbeitung der zur Aktivität gehörigen Token müssen diese jeweils von der Queue entfernt werden, bevor ein Modul für eine innere Aktivität aufgerufen werden kann, oder die Kontrolle an ein aufrufendes Modul zurückgegeben wird. Als Ergebnis wird die eigene Instanz des Moduls zurückgegeben, die den abstrakten Syntaxbaum für die geparssten Token enthält. Die interne Speicherung der nötigen Informationen bleibt dem Implementierer des Moduls überlassen. Die Instanz des Moduls muss nach einem Aufruf der `Parse`–Methode jedoch in der Lage sein, mit der `Compile`–Methode ein der Tokenfolge entsprechendes Petri–Netz zu generieren.

5.1.2 Compiler

In der Folge soll nun der Compiler näher beschrieben werden, der die mit dem API erstellten Module verwendet. Abbildung 26 zeigt die Architektur des Compilers in der Übersicht.

Der Compiler gliedert sich in die Analysephase (*Frontend*) und die Synthesephase (*Backend*). Die Analysephase enthält den *Scanner*, der die lexikalische Analyse durchführt. Der Scanner benutzt einen separaten *Screener*, der zunächst unnötige Informationen aus dem Xaml–Code herausfiltert. Anschließend wird der vom Scanner generierte Tokenstrom vom *Parser* in einen `IActivity`–Syntaxbaum transformiert. In der Synthesephase wird aus diesem zunächst eine Petri–Netz–Datenstruktur erzeugt und anschließend mit deren Ausgabemethoden in ein gewünschtes Ausgabeformat gebracht (vgl. [ALSU07, S.11 f.]).

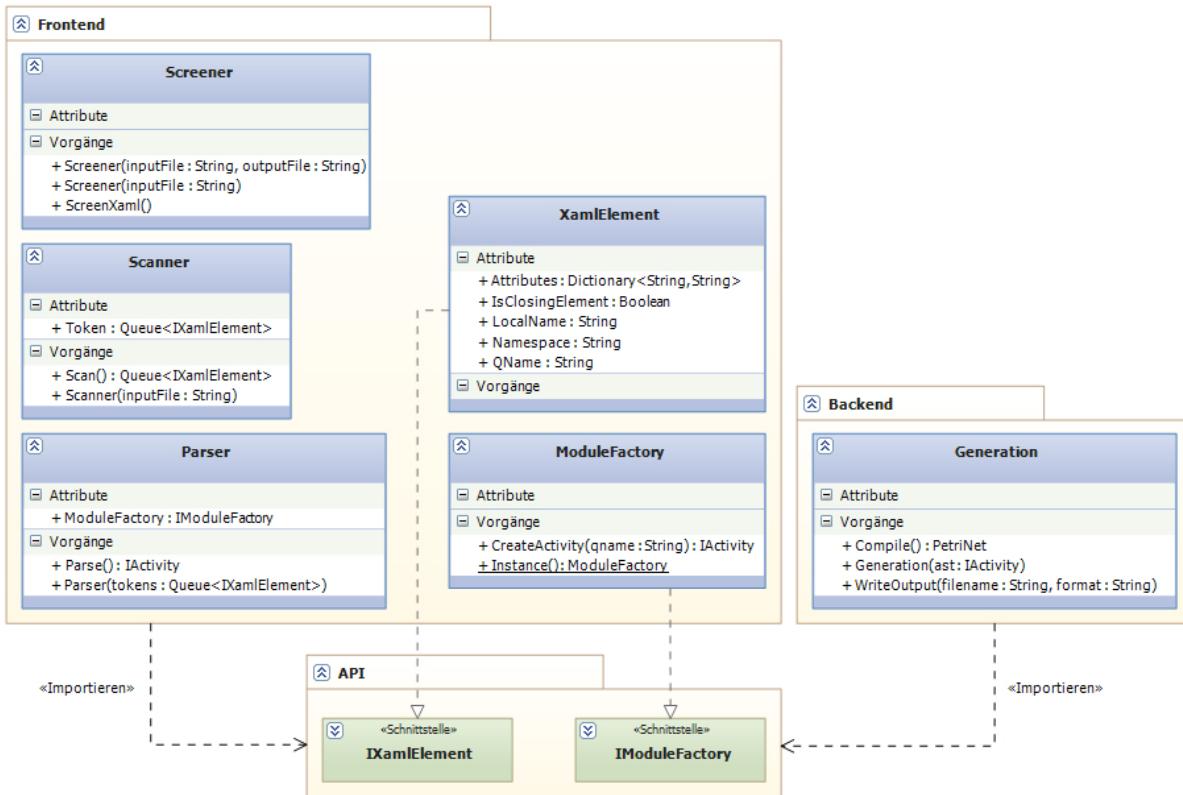


Abbildung 26: Der Compiler WF2oWFN

Screener

Um die Struktur der Xaml– und Xamlx–Dateien zu entschlacken und den Prozess des Parsens zu vereinfachen, werden vor der lexikalischen Analyse im Screener überflüssige Strukturen²¹ entfernt. Da der Screener zur Durchführung seiner Operationen die Xaml in eine *Document Object Model (DOM)*–Struktur einliest, wird zudem die Wohlgeformtheit des Dokuments sichergestellt. Der Xaml–Code wird vor allem durch eine große Menge an Informationen über den visuellen Zustand (*ViewState*) des Designers aufgebläht. Der Screener entfernt deshalb alle zugehörigen Elemente und Attribute aus dem Code. Alle Importe des Workflows werden zudem über Namespace–Deklarationen in der Xaml abgebildet, weswegen viele ungenutzte Namespaces existieren, die ebenso beseitigt werden. Da keine Betrachtung des Datenaspektes durch den Compiler implementiert ist, werden auch alle Datenaspekte, die nicht nötig sind, herausgefiltert. Dazu gehören alle Argument– und Variablen Deklarationen (vgl. [Mic12p]). Der Code verkleinert sich durch diese Maßnahmen auf durchschnittlich 67 % der Originalgröße²². Besonders groß ist die Ersparnis bei Workflows, die noch die ViewState–Informationen beinhalten (36 % der Originalgröße). Aber auch bei Workflows ohne Designer–Informationen kann die Codegröße auf 80 % reduziert werden. Dadurch wird vor allem das Parsen vereinfacht, da keine irrelevanten Daten

²¹Mehr Informationen hierzu unter <http://blogs.msdn.com/b/tilovell/archive/2012/02/15/wf4-xaml-mc-ignoreable-viewstates-hintsizes-visualbasic-settings-and-manipulating-xaml-programmatically.aspx>

²²Basierend auf einer Analyse mit 262 WF–Prozessen aus den WF 4–Samples (<http://www.microsoft.com/en-us/download/details.aspx?id=21459>), WF 3.5–Samples (<http://www.microsoft.com/en-us/download/details.aspx?id=11413>) und den Workflow–Pattern–Implementierungen aus [Len11].

im Tokenstrom erwartet werden müssen. Als positiver Nebeneffekt dieser Transformation ergibt sich zudem eine deutlich verbesserte Lesbarkeit der Prozesse.

Scanner

Die von Screener gesäuberten Xaml(x)–Dateien werden anschließend vom Scanner bearbeitet, der sie für den Parser in einzelne Token zerlegt. Jedes öffnende oder schließende Element des Xaml–Codes stellt hierbei ein Token dar. Da einige Aktivitäten auch Zugriff auf die Elementattribute brauchen, werden diese mit dem Startelement übergeben. Mögliche Elementinhalte, wie Referenzen, werden als dem Element gleichnamiges Attribut hinzugefügt. Das Interface **IXamlElement** definiert den Aufbau eines vom Scanner generierten Tokens. Nach Bearbeitung der gesamten Datei liefert der Scanner eine Queue aller Token zurück.

Parser

Die vom Scanner erzeugte Token–Queue wird an den Parser weitergegeben. Da der Compiler modular aufgebaut ist, ist der Parser letztendlich ein Modul für die Wurzelemente der Xaml(x)–Dateien. Als Grammatik kommt die erarbeitete Spezifikation (siehe Anhang A) zum Einsatz. Mit Kenntnis der zu erwartenden Token kann die Syntax der WF–Aktivitäten mit Hilfe der Queue geparsed werden. Innerhalb des Parsers, bzw. der Module im Allgemeinen, werden die Token nach und nach von der Queue entfernt und analysiert, um nötige Schlüsse für die Generierung aus der Syntax zu ziehen und diese festzuhalten. Die Art und Weise der Speicherung bleibt für den Compiler dabei transparent. Nach Bearbeitung der zur Aktivität gehörigen Token müssen diese jeweils von der Queue entfernt werden, bevor ein Modul für eine innere Aktivität aufgerufen werden kann oder die Kontrolle an ein aufrufendes Modul zurückgegeben wird. Vom Parser aus wird die Parse–Methode der ersten Kindaktivität mit Hilfe der übergebenen **IModulFactory** aufgerufen und sukzessive der abstrakte Syntaxbaum des Prozesses aufgebaut. Wenn die Kontrolle zum Parser zurückkehrt, hält dieser das Wurzelement mit allen Aktivitäten des Prozesses. Dieser Syntaxbaum kann anschließend an das Backend zur Generierung der Petri–Netz–Repräsentation weitergegeben werden.

Generation

Der Syntaxbaum des Parsers wird in der Generierung zum gesamten Petri–Netz zusammengefügt. Die Vorgehensweise ist dabei äquivalent zu der des Parsers. Aus der Generation–Klasse wird die Compile–Methode der ersten **IActivity**–Instanz des Syntaxbaumes aufgerufen und dieser Aufruf in den inneren Modulinstanzen fortgeführt. Jede Instanz erstellt anhand der im vorherigen Schritt abgeleiteten Informationen eine entsprechende Petri–Netz–Repräsentation der Aktivität. Optimierungen der Netzstrukturen obliegen dabei dem Implementierer der Module. Aktuell existieren in fast allen Modulen Optimierungen der Netze anhand von Fallunterscheidungen, die aufgrund der Annotations des Syntaxbaumes der Aktivität getroffen werden. Eine weitergehende Reduktion auf dem Gesamtnett wird aktuell nicht durchgeführt. Nach Abschluss der Methode werden in den jeweiligen Elternmodulen die Netze anhand des Petri–Netz–Interfaces (siehe Abb.8)

verschmolzen. Schlussendlich entsteht dabei das komplette Petri–Netz des WF–Prozesses. Mit Hilfe der Ausgabemethoden der `PetriNet`–Klasse kann diese Datenstruktur dann in ein persistentes Dateiformat überführt werden.

5.2 Entwicklung eines Moduls

Um die angesprochenen Konzepte für einen konkreten Fall zu demonstrieren und die nötigen Schritte zur Erstellung eines eigenen Moduls aufzuzeigen, wird nun die Implementierung eines Moduls ausgeführt. Als Beispiel dient eine strukturierte Aktivität, da sie alle relevanten Aspekte inklusive der Nutzung der `IModuleFactory` veranschaulicht. Eine primitive Aktivität ist dementsprechend einfacher zu implementieren. Als Beispiel wird das Modul der Standardaktivität `While` vorgestellt. Zunächst gilt es die genaue Xaml–Struktur der Aktivität zu kennen. Für die `While`–Aktivität findet sich diese im Anhang A.2.4. Der Xaml–Code wird vom Compiler durch den Screener gesäubert (siehe Kapitel 5.1.2). Alle entfernten Aspekte der Xaml sind demnach auch in der eigenen Implementierung nicht zu beachten. Da `While` eine strukturierte Aktivität ist, implementiert sie das Interface `IComposite` des APIs. Ein Verweis auf das API muss dafür vorher in das Projekt eingebunden werden. Der Übersicht halber wurden in den folgenden Listings standardmäßige Importe und nicht ausschlaggebende Details gekürzt. Listing 6 zeigt zunächst die Basisdefinition des Moduls und die Parse–Methode.

Zunächst müssen die Properties für die Attribute `LocalName`, `QName` und `IModulFactory` aus dem Interface `IComposite` hinzugefügt werden. Da dies trivial ist, wird auf eine Darstellung verzichtet. Die privaten Attribute speichern hierbei die nötigen Informationen der Properties intern. Anschließend kann der Parser der Aktivität implementiert werden. Da der Aufruf des Moduls erfolgt wenn das aktuelle Token mit dem QName der Aktivität übereinstimmt, kann zunächst der öffnende Xaml–Tag von der Queue entfernt werden, weil keine weiteren Informationen daraus gebraucht werden. Anschließend folgt ein optionaler Elementinhalt, da die `While`–Aktivität auch keine Aktivität beinhalten kann. Dies wird durch eine Schleife abgebildet, die überprüft, ob die Queue ein schließendes Element enthält. Innerhalb des Inhaltsbereiches kann die Schleifenbedingung definiert sein. Da diese Informationen für die Petri–Netz–Repräsentation ebenso nicht nötig sind, werden auch diese Token entfernt. Existieren Inhaltselemente auf der Queue, so enthalten diese auch eine auszuführende Aktivität. Die Schleifenbedingung würde andernfalls als Attribut des Wurzelements abgebildet werden. Es wird deshalb mit Hilfe der `IModuleFactory` versucht, für das aktuellen Token eine `IActivity` zu erstellen. Dies geschieht über den Aufruf der Methode `createActivity` und der Übergabe des qualifizierten Namens des Tokens. Wenn die zurückgegebene Aktivität nicht `null` ist, konnte erfolgreich eine Instanz eines Moduls für die Aktivität erstellt werden. Gelang dies nicht, so existiert entweder kein Modul für diese Aktivität oder es ist ein Fehler in der Struktur des Tokenstroms aufgetreten. Dies wird durch das Auslösen einer `ParseException` signalisiert. Auf der Instanz des Moduls wird nun ebenso die Parse–Methode aufgerufen, um die innere Aktivität zu parsen. Kehrt der Aufruf zurück, so kann die `IActivity`–Instanz des Kindelements für den Syntaxbaum intern (Attribut `innerActivity`) gespeichert werden. Nach dem Verlassen der Schleife gilt es noch das schließende Element der Aktivität zu entfernen und seine eigene Instanz an den Aufrufer zurückzugeben.

Listing 6: Parser des While–Moduls

```

1  using WF2oWFN.API;
2  using WF2oWFN.API.Petri;
3
4  namespace WF2oWFN.Modules
5  {
6      partial class While : IComposite
7      {
8          // Properties
9          private readonly String name = "While";
10         private readonly String ns = "http://schemas.microsoft.com/
11             netfx/2009/xaml/activities";
12         private IModuleFactory moduleFactory;
13         // AST
14         private IActivity innerActivity;
15
16         public IActivity Parse(Queue<IXamlElement> token)
17         {
18             //Start Element
19             token.Dequeue();
20             // Optional Body
21             while (!token.Peek().QName.Equals(this.QName) || !token.
22                 Peek().IsClosingElement)
23             {
24                 if (token.Peek().QName.Equals(createQName("While.
25                     Condition")))
26                 {
27                     // Start & End Element
28                     token.Dequeue();
29                     token.Dequeue();
30                 }
31                 // <Activity>
32                 IActivity activity = moduleFactory.CreateActivity(token.
33                     Peek().QName);
34
35                 if (activity != null) {
36                     innerActivity = activity.Parse(token);
37                 }
38                 else {
39                     throw new ParseException(String.Format("No Module
40                         found for activity '{0}'", QName));
41                 }
42             }
43             // End Element
44             token.Dequeue();
45
46             return this;
47         }
48     }
49 }
```

Weiterhin muss das Modul in der Lage sein, mit den in der Parse–Methode generierten Informationen eine Petri–Netz–Repräsentation zu erstellen. Dies geschieht mit der Compile–Methode. Zur Erstellung des Petri–Netzes werden die Klassen aus dem API benutzt. Der Aufbau des gesamten Petri–Netzes erfolgt ähnlich zur Erstellung des Syntaxbaumes. Dabei wird zwischen den Aufrufen jeweils das gemeinsame Stammnetz (Parameter *phylum*) weitergegeben, in das die Teilnetze eingefügt und miteinander verbunden werden. Zunächst gilt es sicherzustellen, dass alle Namen der Stellen und Transitionen im Gesamtnetz eindeutig sind. Dies geschieht über die Konvention der Nutzung der Eigenschaft *ActivityCount*. Dieser Zähler enthält eine eindeutige ID der aktuellen Instanz. Vor Aufruf einer neuen Instanz muss dieser Zähler erhöht werden. Damit ist sichergestellt,

dass keine Benennungskonflikte zwischen den Teilnetzen entstehen. Da die Teilnetze in den übergeordneten Aktivitäten wieder verschmolzen werden, müssen alle Teilnetze das in Abb.8 definierte Petri–Netz–Interface besitzen. Für die Benennung der Interfacestellen gilt die folgende Konvention:

`$ActivitiyCount.internal.initialized` und `$ActivitiyCount.internal.closed`

Die Namensgebung aller anderen Stellen kann frei gewählt werden, solange sie den eindeutigen *ActivityCount* nutzen. Listing 7 zeigt die Implementierung der Petri–Netz–Generierung. Zum besseren Verständnis empfiehlt es sich erneut das entsprechende Pattern in Abb.16a zu betrachten. Wenn die beim Parsen gespeicherte innere Aktivität nicht leer ist, so wird auf dieser ebenfalls die *Compile*–Methode aufgerufen. Ansonsten findet eine Optimierung des Netzes statt. Nachdem der Aufruf der *Compile*–Methode der inneren Aktivität beendet ist, werden die beiden Teilnetze im Wurzelbaum miteinander verbunden. Hierzu wird eine Verbindung zur initialen Stelle der Aktivität erstellt und die finale Stelle der Aktivität mit der initialen der *While*–Aktivität verschmolzen. Zum Schluss wird der Wurzelbaum wieder an den Aufrufer des Moduls zurückgeben, um übergeordnete Merge–Vorgänge zu ermöglichen.

Listing 7: Generierung des *While*–Moduls

```

1 partial class While : IComposite
2 {
3     public PetriNet Compile(PetriNet phylum)
4     {
5         String prefix = phylum.ActivityCount + ".internal.";
6         // Interface
7         Place p1 = phylum.newPlace(prefix + "initialized");
8         Place p2 = phylum.newPlace(prefix + "closed");
9         // Inner Petri Net
10        Place condition = phylum.newPlace(prefix + "condition");
11        Transition evalCondition = phylum.newTransition(prefix + "evalcondition");
12        Transition init = phylum.newTransition(prefix + "start");
13        Transition end = phylum.newTransition(prefix + "end");
14        phylum.newArc(p1, evalCondition);
15        phylum.newArc(evalCondition, condition);
16        phylum.newArc(condition, init);
17        phylum.newArc(condition, end);
18        phylum.newArc(end, p2);
19
20        // Inner Activity
21        if (innerActivity != null) {
22            // New Activity
23            phylum.ActivityCount += 1;
24            int currentID = phylum.ActivityCount;
25            // Compile
26            innerActivity.Compile(phylum);
27            // Connect & Merge
28            phylum.newArc(init, currentID + ".internal.initialized");
29            phylum.Merge(p1, currentID + ".internal.closed");
30        }
31        else {
32            // Empty
33            phylum.newArc(init, p1);
34        }
35
36        return phylum;
37    }
38 }
```

Um das entwickelte Modul für den Compiler nutzbar zu machen, muss es zu einer Klassenbibliothek (*.dll) kompiliert werden und in den Unterordner MODULES des Compilers kopiert werden. Die Instanz der **IModuleFactory** lädt und verwaltet beim Programmstart des Compilers alle von **IActivity** abgeleiteten Assemblies des Ordners.

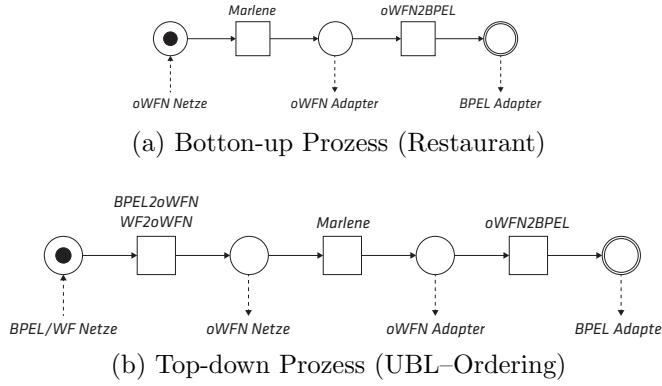


Abbildung 27: Abläufe, Artefakte und Tools der Integrationsprozesse

6 Fallbeispiele

Mit Hilfe des vorgestellten Prototypen und den bestehenden Tools soll nun anhand von zwei Fallbeispielen die praktische Anwendbarkeit einer teilautomatisierten Integration zwischen BPEL- und WF-basierten Prozesssystemen evaluiert werden. Dabei liegt der Fokus auf der Verhaltenskorrektheit der komponierten Systeme. Darüber hinaus werden weitere beobachtete Interoperabilitäts- und Kommunikationsprobleme der Systeme und mögliche Lösungen dafür thematisiert. Als Entwicklungsumgebung kommt im Falle von WF das Visual Studio 2010 Ultimate SP1 mit dem .NET Framework 4 Platform Update 1 zum Einsatz. Die Laufzeitumgebung für die Prozesse stellt der integrierte ASP.NET Development Server bereit. Die BPEL-Prozesse werden von Apache ODE 1.3.5 ausgeführt, das von einem Tomcat 7.0.27 Server gehostet wird²³. Um sowohl die Integration bestehender heterogener Prozesssysteme zu zeigen als auch die grundlegenden Aspekte der automatisierten Petri-Netz-Integration und der Interaktion zwischen den Systemen über Webservices zu untersuchen, wird im Folgenden sowohl eine Bottom-up als auch eine Top-down Integration betrachtet. Im Falle des Bottom-up Ansatzes werden deshalb zwei Services anhand einer gegebenen Petri-Netz-Struktur modelliert und integriert, um die Funktionsfähigkeit der Tools und die Interoperabilität zwischen den resultierenden Services zu evaluieren. Da WF sämtliche Kommunikation über WCF abwickelt, sind die folgenden technischen Details somit nicht nur für WF gültig, sondern für alle Programme innerhalb des .NET-Frameworks, welche WCF für ihre Kommunikation nutzen. Beim Top-down Ansatz sind zwei Services gegeben. Zunächst wird Tool-basiert deren Petri-Netz-Repräsentation abgeleitet, um ebenfalls anschließend für beide einen entsprechenden Adapter zu generieren und zu implementieren. Für die Bottom-up Betrachtung wird auf das Restaurant-Szenario aus Kapitel 3.3 zurückgegriffen, während für den Top-Down Fall eine konkrete Implementierung des Ordering-Prozesses aus der UBL 2.1 Spezifikation [OAS11] untersucht wird. Da der Fokus der Arbeit auf WF und der Petri-Netz-Integration liegt, sind die Ausführungen bei diesen Themen entsprechend detailliert. Die Betrachtung der BPEL-Prozesse wird dagegen knapper gehalten. Bei Unklarheiten empfiehlt es sich, die angegebenen weiterführenden Quellen zu konsultieren. Abbildung 27 zeigt den Ablauf, die Artefakte und die eingesetzten Tools der Integrationen in der Übersicht.

²³Eine genaue Installationsbeschreibung und die Bezugsquellen aller nötigen Software befindet sich im Ordner SOFTWARE auf der beiliegenden CD.

6.1 Restaurant–Prozess

Die Abbildungen 5a und 5c in Kapitel 3.3 zeigen die Petri–Netze der beiden Parteien Koch und Tourist. Wie bekannt ist, sind die beiden a priori nicht miteinander kompatibel, da der Koch eine Bestellung erwartet und zudem mit dem Kochen aufhört, sobald er eine frühzeitige Entlohnung erhält. Der dort dargestellte idealtypische Adapter wird vernachlässigt, da die tatsächliche Ausgabe der automatisierten Adaptersynthese betrachtet werden soll. Zunächst sollen jedoch die beiden Ausgangsnetze in äquivalente Webservice–basierte Prozesse umgesetzt werden. Der Koch soll durch einen BPEL–Prozess implementiert werden, während der Tourist als WF–Prozess modelliert wird.

6.1.1 Tourist

Mit Kenntnis der Pattern aus Kapitel 4 lässt sich die Struktur eines entsprechenden WF–Prozesses für den Touristen einfach an dessen Petri–Netz erkennen. In diesem Fall ist es eine Sequenz aus einer `Send`– und einer `Receive`–Aktivität, für die Geldübergabe und für den Essensempfang. Abbildung 28 zeigt den gesamten WF–Prozess. Die beiden zusätzlichen Aktivitäten `Init` und `Close` implementieren ein Request/Response–Pattern über `ReceiveAndSendReply`–Aktivitäten, um den Prozess zu initiieren und ein Ergebnis an den Aufrufer zurückzusenden. Die `Init`–Aktivität ist nötig, da die Instanzierung eines Prozesses grundsätzlich durch den Empfang einer Nachricht angestoßen werden muss. Die Eigenschaft `CanCreateInstance` der `Init`–Aktivität ist dafür gesetzt. Die beiden Aktivitäten dienen einzlig und allein zum Starten und Testen der Komposition, sind vom entscheidenden durch das Petri–Netz modellierten Kontrollfluss jedoch entkoppelt.

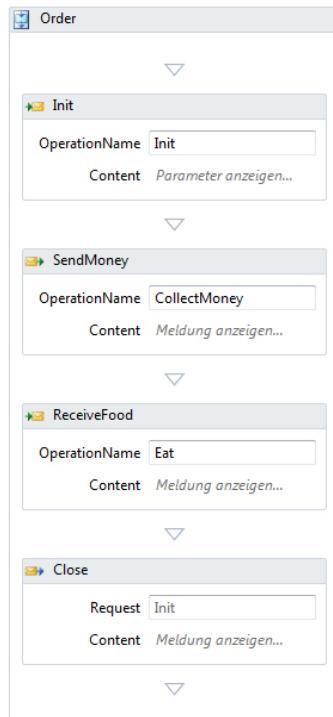


Abbildung 28: WF–Prozess Tourist

Um nun mit der **Send**-Aktivität des Touristen einen externen Webservice aus WF heraus aufzurufen, gibt es zwei Möglichkeiten. Der einfache und automatisierte Weg führt über einen sog. *Dienstverweis*. Dieser kann im Kontextmenü des Unterordners Verweise in der Projektmappe hinzugefügt werden. Durch Angabe der Adresse des Services, kann nach Auswahl eines Dienstvertrages ein entsprechender Dienstverweis generiert werden. Nach einem erneuten Erstellen des Projektes ist für jede im Dienstvertrag erkannte Operation eine eigene **CustomActivity** verfügbar, in deren Eigenschaften Referenzen zu den Übergabennachrichten definiert werden können. Hierfür wurden automatisch gleichlautende Proxy-Klassen erzeugt. Alle nötigen Bindings und Endpoints werden ebenfalls automatisiert in die Konfiguration geschrieben (vgl. [Mic12a]). Wenn man nicht über **CustomActivities** auf Dienste zugreifen möchte, kann man das gleiche Ergebnis auch durch eine manuelle Konfiguration der Aktivitäten und Übergabennachrichten erzielen. Grund hierfür könnte beispielsweise eine flexiblere, besser wartbare Konfiguration sein, welche man selbst beeinflussen möchte. Weiterhin müsste für die Übersetzung mit WF2oWFN aktuell für jede generierte **CustomActivity** jeweils ein eigenes Modul geschrieben werden, da der Compiler noch keine Module unterstützt, die mehrere Petri-Netz-äquivalente Aktivitäten bearbeiten können. Zudem besitzen die Aktivitäten jeweils ihren eigenen Namen samt Namespace, obwohl sie nur **Receive**-, **Send**-Aktivitäten oder Kombinationen daraus sind, was eine überschaubare Bündelung gleichartiger Konstrukte erschwert. Deshalb bietet es sich für die Untersuchung an, die Basisaktivitäten zu nutzen, um eine effiziente Automatisierung der Transformation zu erreichen. Hierfür kann man sich einer ähnlichen Vorgehensweise bedienen, die anschließend zur Definition des Dienstvertrages des eigenen Services nötig ist. Damit der Service in von ihm erwarteter Weise angesprochen wird, müssen in den Eigenschaften der **Send**-Aktivität zunächst ein gültiger Dienstvertrag (*ServiceContractName*), Operationsaufruf (*OperationName*), das Binding und die Endpointadresse des Services angegeben werden. Zunächst ist es nötig, den Dienstvertrag des entfernten Services zu definieren. Das Pattern hierfür besteht aus dem geklammerten Namespace und dem Namen des gewünschten **portTypes**.

{ http://uniba.de } Guide

Anschließend kann aus dem gewählten Dienstvertrag eine aufzurufende Methode und die Adresse des Webservices in den Eigenschaften angegeben werden. Wenn keine besonderen Anforderungen an die Kommunikation gestellt werden, empfiehlt es sich bei der Auswahl des Bindings auf das **BasicHttpBinding** zurückzugreifen, da es die größtmögliche Interoperabilität gewährleistet²⁴. Damit die Interaktion mit dem Service korrekt vonstatten gehen kann, gilt es zuletzt die strukturell erwarteten Nachrichtentypen zu übertragen. Da im konkreten Fall noch keine definiert wurden, sollen sie nun in WF definiert werden.

Zur Definition der Inhaltstruktur eignen in diesem Fall am besten mit **DataContract** annotierte Klassen. Zunächst wird dafür die Klassendeklaration mit dem **DataContract**-Attribut gekennzeichnet. Um die entsprechenden Attribute verwenden zu können muss manuell ein Verweis auf die **System.Runtime.Serialization** Assembly in das Projekt eingebunden werden. Einzelne Felder oder Eigenschaften der Klasse, die in der Nachricht verwendet werden sollen, müssen mit dem **DataMember**-Attribut markiert werden (vgl. [Buk10, S.322 f.]). Abb.8 zeigt die Klasse **Money**, die für den Geldtransfer genutzt werden soll.

²⁴Das **BasicHttpBinding** ist kompatibel zum WS-I Basic Profile 1.1 (vgl. [Mic12c])

Listing 8: Nachrichtendefinition Money

```

1 [DataContract(Namespace="http://uniba.de")]
2 public class Money
3 {
4     [DataMember(Order=0)]
5     public int user;
6     [DataMember(Order=1)]
7     public int amount;
8 }
```

Durch die Eigenschaft *Namespace* wird der Namensraum der Nachricht definiert. Da wir die gleiche Nachrichtendefinition für alle Services benutzen wollen, können wir hiermit einen bestimmten Namespace für die Nachricht und deren Elemente erzwingen. Generell werden die Nachrichtenelemente von WF bei der Übertragung in alphabetischer Reihenfolge ihrer Bezeichnung erwartet (d.h. *amount* vor *user*), egal in welcher Reihenfolge sie definiert wurden (vgl. [Mic12d]). Dieses Verhalten provoziert Interoperabilitätsprobleme mit anderen Webservices, die ihre Nachrichten standardmäßig nach der Definitionsreihenfolge serialisieren. In diesem Fall würde eine **SerializationException** von WF ausgelöst werden. Um dies zu vermeiden, kann durch Angabe der Eigenschaft *Order* die Reihenfolge der Serialisierung und Deserialisierung der Felder beeinflusst werden. Nicht nur hier zeigt sich die Stärke des Dienstverweises, der dem Nutzer diese fehleranfällige Arbeit abnimmt. Abbildung 9 zeigt die von WF aus dem **DataContract** generierte XSD, die für die Webservice-Kommunikation genutzt wird und auch in den folgenden BPEL-Services verwendet werden soll. Die weiteren nötigen Nachrichten **Food** und **Order** bestehen ebenfalls aus einem Feld zur Nutzeridentifikation und einem weiteren Feld mit einer Zeichenfolge für die Speise.

Listing 9: XSD Money

```

1 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
2   targetNamespace="http://uniba.de">
3   <xs:complexType name="Money">
4     <xs:sequence>
5       <xs:element name="user" type="xs:int"/>
6       <xs:element name="amount" type="xs:int"/>
7     </xs:sequence>
8   </xs:complexType>
9 </xsd:schema>
```

Als nächstes gilt es die **Receive**-Aktivität und damit den eigenen Dienstvertrag des Services zu definieren. Zunächst muss dafür der *ServiceContractName* der Aktivität wie bei der Konfiguration der **Send**-Aktivität mit dem geklammerten Namespace und dem gewünschten Namen des **portTypes** in den Eigenschaften definiert werden. Alle unterschiedlichen Werte erhalten einen eigenen **portType** in der finalen WSDL. Hierdurch ist eine logische Trennung verschiedener Dienstverträge möglich. Der **ServiceContract** des Workflows wird automatisch aus allen vorhandenen **Receive**-Aktivitäten erstellt. Es existiert aktuell keine Möglichkeit ihn über Interfaces zu definieren (vgl. [Buk10, S.324])²⁵.

Da die Aufrufe zwischen den beteiligten Services asynchron erfolgen und mehrere Instanzen eines Services gleichzeitig aktiv sein können, muss eine Korrelation stattfinden, die die Nachrichten zu der korrekten Instanz eines Services weiterleitet. Da Microsoft für

²⁵Ab der kommenden Version 4.5 wird dies unterstützt. Siehe [http://msdn.microsoft.com/en-us/library/hh305676\(v=vs.110\)](http://msdn.microsoft.com/en-us/library/hh305676(v=vs.110))

die kontextbasierte Korrelation in .NET ein eigenes Protokoll [Mic12h] nutzt, kommt es hier erwartungsgemäß zu Interoperabilitätsproblemen. Deswegen wird eine inhaltsbasierte Korrelation verwendet (siehe Kapitel 2.1.3). Alle bisher definierten Nachrichten enthalten dafür ein Feld für eine Nutzeridentifikation. Dieser Wert soll eindeutig sein und erlaubt damit eine eindeutige Zuordnung der Nachrichten. Um den Service schlussendlich zu veröffentlichen, genügt es das Projekt im Debug-Modus auszuführen, was den ASP.NET Development Server startet, der den Service ausführt.

6.1.2 Koch

Das Petri-Netz des Kochs aus Abb.5c lässt sich z.B. wie der Prozess in Abb.29 abbilden. Nach dem Empfang einer Bestellung beginnt der Koch mit der Arbeit. Falls er vor Ende des Vorgangs eine Bezahlung erhält, modelliert durch einen *onMessage*-Zweig in einer Pick-Aktivität, hört er auf zu kochen und behält das Geld ohne eine Leistung zu erbringen. Nach einer gewissen Zeit, modelliert durch eine *onAlarm*-Zeitspanne, gibt er ansonsten das fertige Essen aus und erwartet die Bezahlung.

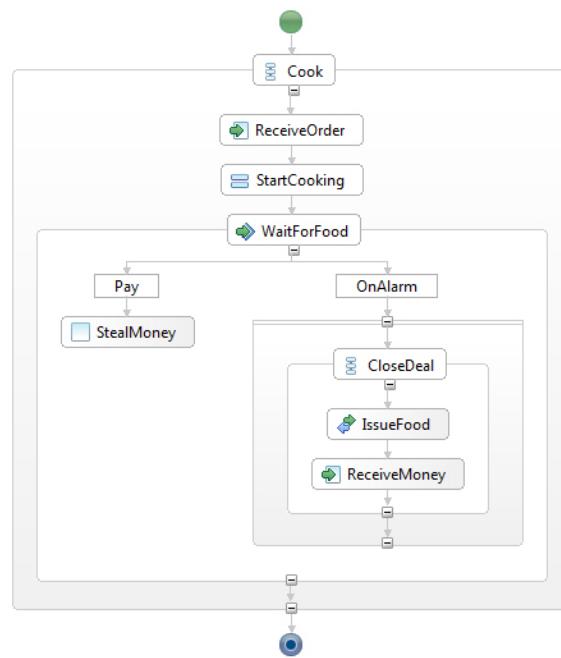


Abbildung 29: BPEL-Prozess Koch

Die Datentypen für den Prozess werden aus den von WF generierten XSDs übernommen (siehe Listing 9). Kämen hier syntaktisch andere Nachrichtentypen zum Einsatz, müsste diese zusätzliche Inkompatibilität später im Adapter überwunden werden. Da dies jedoch mit einfachen Zuweisungsoperationen vom einen zum anderen Typ gelöst werden kann und somit außer vielen Kopieroperationen trivial ist, wird darauf verzichtet. Im Prozess müssen zwei `partnerLinks` definiert werden. Einen, der auf den `portType` seiner eigenen Rolle verweist und die beiden Operationen für die Bestellannahme und die Bezahlung anbietet und einen der den `portType` des Partners referenziert, der eine Methode für die Essensausgabe enthält. In den Receive- und Invoke-Aktivitäten des Services können dann die entsprechenden `partnerLinks` und die gewünschte Operation ausgewählt werden. Die WSDL des Services wird vom Tool größtenteils selbstständig aus dem vorhandenen Code

abgeleitet. Anpassungen können zudem auch über den interaktiven Designer vorgenommen werden. Genauso wie beim Touristen, wird die Nachrichtenzuordnung durch eine inhaltsbasierte Korrelation auf Basis der Nutzeridentifikation durchgeführt. Wie ebenso in Kapitel 2.1.2 beschrieben, muss zuletzt noch ein Deployment Descriptor für Apache ODE definiert werden. Um den Prozess zu deployen, genügt es den Ordner mit allen Artefakten in den Unterordner WEB-INF/PROCESSES der ODE-Installation zu kopieren.

6.1.3 Adaptersynthese

Wie bereits gezeigt, sind die beiden Services noch nicht miteinander kompatibel. Mit Hilfe des Tools Marlene soll nun zunächst ein Petri-Netz-Adapter für die beiden Partner generiert werden. Um die Semantik des Adapters zu beeinflussen, müssen entsprechende Adapterregeln definiert werden. Hier wird auf die schon definierte SEA zurückgegriffen (siehe Tabelle 1). Standardmäßig wird von Marlene der liberalste Adapter generiert, d.h. der Adapter der die wenigstens Restriktionen in Bezug auf den Kontrollfluss an das Netz legt. Als Kompatibilitätskriterium wird die Deadlock-Freiheit gefordert. Alle Parameter können bei Bedarf entsprechend verändert werden. Um den liberalsten Adapter für zwei Netze unter den gegebenen Standardeinstellungen zu generieren, genügt der folgende Programmaufruf:

```
marlene tourist.owfn cook.owfn -o adapter.owfn -r rulefile.ar
```

Abbildung 30 zeigt den generierten liberalsten Adapter für den Restaurant-Prozess.

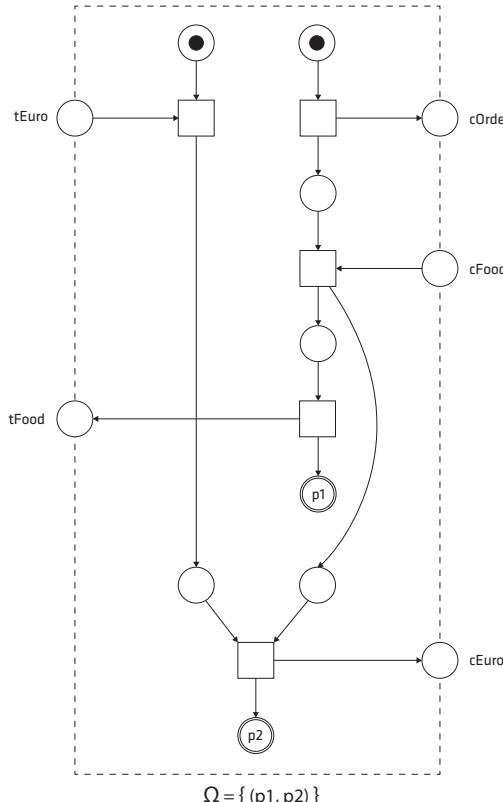


Abbildung 30: Liberalster Adapter Restaurant

Ein beliebiger Adapter stellt einen Ausschnitt des liberalsten Adapters dar, der einige Restriktionen an den Kontrollfluss setzt. Ein solcher kann mit Marlene durch Konfiguration des Parameters $-t$ erzeugt werden:

```
marlene tourist.owfn cook.owfn -t arbitrary -o adapter.owfn -r rulefile.ar
```

Der Adapter in Abb.31a kann im Vergleich zum liberalsten Adapter nur durch einen Teil dessen möglicher Strategien bedient werden. Der Adapter erwartet beispielsweise zunächst die Bezahlung des Touristen, bevor er eine Bestellung beim Koch aufgibt. Diese Reihenfolge lässt der liberalste Adapter offen. Aufgrund der getroffenen Einschränkungen und der damit verbundenen geringeren Nebenläufigkeit, verkleinert sich das Netz des beliebigen Adapters. Nicht nur wegen des kleineren Netzes, sondern auch weil das Tool *oWFN2BPEL*²⁶, mit dem das Petri–Netz in einen BPEL–Prozess übersetzt werden soll, größere Probleme bei der Transformation aufweist, wird der beliebige Adapter als Ausgangspunkt für die Übersetzung nach BPEL genutzt. Mehrere Versuche, andere Adapternetze als das gegebene zu transformieren, scheiterten an offensichtlich programmbedingten Laufzeitfehlern. Diverse Quellen erwecken dabei den Eindruck einer gewissen Reife. V.d.Aalst et al. [VMSW09] sprechen von dem Tool oWFN2BPEL, das Open Nets in abstrakte BPEL–Prozesse übersetzt und zusammen mit anderen Tools die Prozesskette zwischen BPEL und Open Nets komplettiert. Auch von Lohmann und Kleine [LK08] wird es als verbindendes Glied bezeichnet, das das Framework zur Synthesierung von BPEL–Prozessen vervollständigt und die vollautomatische Partnersynthese für einen gegebenen BPEL–Prozess ermöglicht. Es wird dabei sogar von einer einfachen Integration des Frameworks in gewerbliche BPEL–Design–Tools gesprochen. Die interne Bugreport Seite²⁷ zeigt jedoch, dass die angesprochenen Probleme offensichtlich bekannt sind und eine Neuimplementierung vorgeschlagen wird. Da das Tool zudem seinen eigenen oWFN–Parser implementiert, der nicht die gesamte oWFN–BNF [Loh10, S.10] unterstützt, muss auch das Netz zuerst manuell angepasst werden. Die aktuellste Version von Marlene modelliert das Petri–Netz mit einer Interface–Deklaration und einer finalen Bedingung. oWFN2BPEL hingegen erlaubt nur eine Aufteilung der Stellen in Internal–, Input– und Output–Stellen. Weiterhin wird nur eine Menge an finalen Markierungen unterstützt und keine finale Bedingung. Der boolesche Ausdruck muss deswegen in eine Menge mit finalen Markierungen transformiert werden. Listing 10 zeigt die Ausgabe des Tools Marlene und Listing 11 zeigt die transformierte Version, die von oWFN2BPEL akzeptiert wird.

Listing 10: Marlene oWFN–Format

```

1 INTERFACE
2 PORT tourist
3   INPUT    tMoney;
4   OUTPUT   tFood;
5 PORT cook
6   INPUT    cFood;
7   OUTPUT   cOrder, cMoney;
8
9 PLACE p0, (...);
10
11 FINALCONDITION
12 (controller.p3 = 1) AND ALL_OTHER_PLACES_EMPTY;
```

²⁶Eine kompilierte Version des Tools findet sich im Ordner SOFTWARE\oWFN2BPEL auf der CD. Dokumentation und Download unter <http://download.gna.org/service-tech/owfn2bpel/>

²⁷Siehe <http://gna.org/bugs/?12447>

Listing 11: oWFN2BPEL oWFN-Format

```

1 PLACE
2   INTERNAL p0, (...);
3   INPUT    tMoney, cFood;
4   OUTPUT   tFood, cOrder, cMoney;
5
6 FINALMARKING controller.p3 : 1;

```

Nichtsdestotrotz kann die Prozesskette für den konkreten Adapter durchgeführt werden. Unabhängig davon ist der erzeugte Petri-Netz-Adapter auch fähig, bei einer manuellen Übersetzung hilfreich zu sein. Um einen validen Adapter zu implementieren, kann z.B. ein möglicher Weg des Netzes nachvollzogen werden. Der folgende Aufruf transformiert im Erfolgsfall ein Petri-Netz in einen abstrakten BPEL-Prozess:

```
owfn2bpel -i adapter.owfn -o adapter.bpel
```

Listing 12 zeigt den generierten abstrakten BPEL-Prozess.

Listing 12: Abstrakter BPEL-Adapter

```

1 <process name="Act_process_adapter.owfn"
2   targetNamespace="http://docs.oasis-open.org/wsbpel/2.0
3     /process/abstract"
4   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/abstract"
5   suppressJoinFailure="yes"
6   xmlns:template="http://docs.oasisopen.org/wsbpel/2.0
7     /process/abstract/simple-template/2006/08"
8   abstractProcessProfile="http://docs.oasisopen.org/wsbpel/2.0
9     /process/abstract/simple-template/2006/08">
10
11 <partnerLinks>
12   <partnerLink name="generic_pl" partnerLinkType="##opaque"
13     myRole="##opaque" partnerRole="##opaque" />
14 </partnerLinks>
15
16 <variables>
17   <variable name="Var_tEuro" element="##opaque" />
18   <variable name="Var_cFood" element="##opaque" />
19   <variable name="Var_tFood" element="##opaque" />
20   <variable name="Var_cOrder" element="##opaque" />
21   <variable name="Var_cEuro" element="##opaque" />
22 </variables>
23
24 <sequence name="Act_sequence">
25   <receive name="Act_tEuro" partnerLink="generic_pl"
26     operation="tEuro" variable="Var_tEuro" />
27
28   <flow name="Act_t1">
29     <sequence name="Act_sequence">
30       <invoke name="Act_cOrder" partnerLink="generic_pl"

```

```

31      <invoke name="Act_tFood" partnerLink="generic_pl"
32          operation="tFood" inputVariable="Var_tFood" />
33    </sequence>
34
35    <opaqueActivity name="Act_engine.tEuro_int" />
36  </flow>
37
38  <invoke name="Act_cEuro" partnerLink="generic_pl"
39      operation="cEuro" inputVariable="Var_cEuro" />
40  </sequence>
41</process>

```

Um diesen abstrakten Prozess zu einem ausführbaren zu vervollständigen, sind einige Schritte nötig. Damit die Veränderungen die Kompatibilität des abstrakten Prozesses nicht verändern, sind in [OAS07, S.151 ff.] alle konsistenzerhaltenden Veränderungsmöglichkeiten hin zu einem ausführbaren BPEL-Prozess definiert. Zunächst müssen die Namespaces angepasst werden. Hierzu wird der Standardnamespace auf den des ausführbaren WS-BPEL Prozesses geändert und die abstrakten Profile-Templates gelöscht. Der *target-Namespace* wird auf einen eigenen Namenspace festgelegt. Per Importanweisung werden die WSDLs der beiden Partner, inklusive XSDs, mit den definierten Nachrichtentypen importiert (vgl. [OAS07, S.32 f.]). Da das oWFN für das Tool ohne Interfaces definiert wurde, fehlen **partnerLinks**, die noch deklariert werden müssen. Für jeden Partner-Service wird ein **partnerLink** erstellt, der jeweils auf die **partnerLinkTypes** verweist, die in den WSDLs definiert sind. Im Falle des WF-Partners wird in die importierte WSDL noch ein passender **partnerLinkType** eingefügt. Der ersten Aktivität wird das Attribut *createInstance* zugewiesen, um die Prozessinstanz zu initialisieren. Die Variablen werden jeweils auf die korrekten Elementtypen abgebildet. Für jede **receive-** und **invoke-**Aktivität müssen die korrekten **partnerLinks** gesetzt und eine entsprechende Operation ausgewählt werden. Die **opaqueActivity** im parallelen Zweig zur Essenszubereitung, wird durch ein **assign** ersetzt, welches das Geld des Touristen auf die vorgesehene Variable des Kochs transferiert, bis die Essensaushabe stattgefunden hat. Zur Variableninitialisierung und zum Transfer werden zwei weitere **assign**-Aktivitäten eingefügt. Keine der bestehenden Aktivitäten wird verändert oder gelöscht, so wie es durch die Richtlinien vorgesehen ist. Anschließend wird genauso wie in Kapitel 6.1.2 die Korrelation konfiguriert. Insgesamt sollten zudem die Benennungen in aussagekräftigere Namen verändert werden, was jedoch nur Veränderungen kosmetischer Natur sind.

Abbildung 31 stellt das Netz des Adapters und sein BPEL-Äquivalent gegenüber. So lässt sich die sich gleichende Struktur der beiden Repräsentationen erkennen. Nach dem initialen Nachrichtenempfang spaltet sich der Kontrollfluss auf, was in BPEL von einem **flow**-Konstrukt modelliert wird. Der eine Zweig sendet die Bestellung, empfängt das Essen und vereinigt sich anschließend wieder mit dem anderen Zweig, der das Geld aufbewahrt hat, um die Bezahlung durchzuführen und den Prozess abzuschließen²⁸.

Wegen der Probleme mit oWFN2BPEL wurde neben dem BPEL-Adapter zudem manuell ein weiterer Adapter in WF für die beiden Netze implementiert. Er unterscheidet sich im Wesentlichen nur durch eine etwas andere Abfolge des Nachrichtenaustausches, weswegen auf eine explizite Darstellung verzichtet wird²⁹.

²⁸Das vorgestellte Fallbeispiel befindet sich auf dem beiliegenden Datenträger im Ordner CASES\BPEL-RESTAURANT

²⁹Die gesamte ausführbare Orchestration befindet sich im Ordner CASES\WF-RESTAURANT

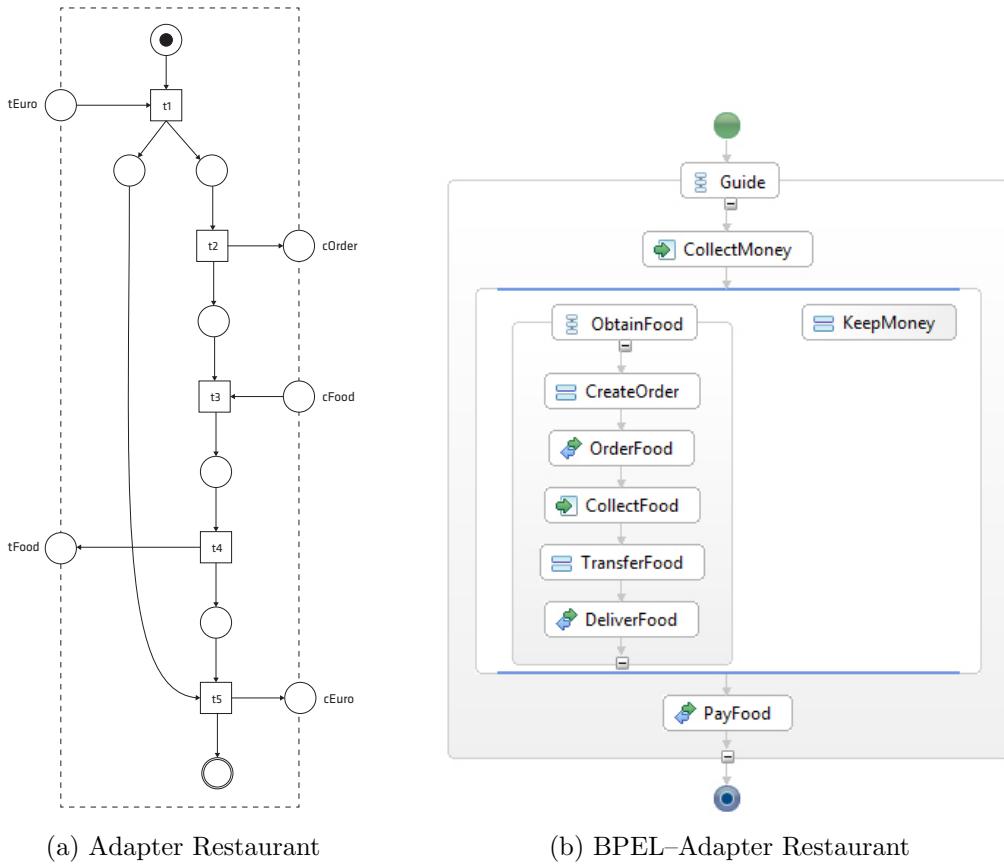


Abbildung 31: oWFN- und BPEL-Adapter Restaurant

6.1.4 Integration

Da nun Implementierungen für alle Parteien bereitstehen, gilt es zu beobachten, ob die Komposition aller harmoniert. Durch die Generierung des Adapternetzes wurde implizit schon die Steuerbarkeit des Gesamtnetzes überprüft. Der Kontrollfluss beinhaltet somit keine möglichen Deadlocks mehr. Wie sich herausstellt, existieren jedoch Probleme in der Webservice-Interoperabilität der verwendeten Systeme. Bei einem Aufruf des Tourist-Services verweigert der ASP.NET Development Server die Antwort des Adapters mit einem *Bad Request (400)*. Laut [W3C99, S.64] deutet dies auf eine nicht verstandene Syntax der Nachricht hin. Listing 13 zeigt die von Apache ODE gesendete Nachricht.

Listing 13: Apache ODE Nachrichtenkodierung

```

1 POST /Service.xamlx HTTP/1.1
2 Content-Type: text/xml; charset=UTF-8
3 SOAPAction: "http://uniba.de/Table/ServeFood"
4 User-Agent: Axis2
5 Host: 127.0.0.1:8888
6 Transfer-Encoding: chunked
7
8 317
9 <?xml version='1.0' encoding='UTF-8'?>
10 <Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
11   <Header> (...) </Header> <Body> (...) </Body>
12 </Envelope>
13 0

```

Wie zu erkennen ist, wurde die Nachricht mit dem sog. *Chunked Transfer Coding* übertragen. Dabei wird die gesamte Nachricht durch eine Serie von einzelnen Teilstücken übermittelt, die jeweils am Beginn mit ihrer Länge gekennzeichnet sind. Wenn das Ende der Nachrichtenserie erreicht ist, indiziert durch eine Länge von null, kann die Nachricht zusammengefügt werden (vgl. [W3C99, S.24 f.]). Ein Blick auf Listing 14 zeigt, dass WF standardmäßig, mit Angabe der Datenlänge, die gesamte Nachricht auf einmal versendet. Nach einigen Tests stellt sich zudem heraus, dass WF Antworten mit Chunked Transfer Coding akzeptiert, lediglich keine Anfragen. Warum dies so ist, kann nicht abschließend geklärt werden, es liegt jedoch nahe, dass damit Sicherheitsfragen verbunden sind, um z.B. *Denial of Service (DoS)*-Attacken zur Überlastung des Servers durch dauerhafte Anfragen zu verhindern.

Listing 14: WF Nachrichtenkodierung

```

1 POST /ode/processes/fastfood HTTP/1.1
2 Content-Type: text/xml; charset=utf-8
3 SOAPAction: "http://uniba.de/fastfoodPortType/Order"
4 Host: 127.0.0.1:8888
5 Content-Length: 225
6 Expect: 100-continue
7 Accept-Encoding: gzip, deflate
8 Connection: Keep-Alive
9
10<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
11    <Header> (...) </Header> <Body> (...) </Body>
12</Envelope>
```

Um Apache ODE dazu zu bringen die gleiche Codierung wie WF zu benutzen, kann man im Installationsverzeichnis die Konfigurationsdatei WEB-INF\CONF\AXIS2.XML editieren. Hierzu existiert für jedes Protokoll ein entsprechender Eintrag, in dem die Nachrichtenkodierung (*Transfer-Encoding*) konfiguriert werden kann. Um zu erreichen, dass die Nachrichten komplett übertragen werden, genügt es die entsprechende Zeile auszukommentieren (siehe Listing 15).

Listing 15: Axis2 Http-Konfiguration

```

1 <transportSender name="http"
2   class="org.apache.axis2.transport.http.CommonsHTTPTransportSender">
3     <parameter name="PROTOCOL" locked="false">
4       HTTP/1.1
5     </parameter>
6     <!--
7       <parameter name="Transfer-Encoding" locked="false">
8         chunked
9       </parameter>
10      -->
11 </transportSender>
```

Nach dieser Änderung werden schlussendlich auch alle Nachrichten korrekt zwischen den beteiligten Services versendet und die Integration ist erfolgreich durchgeführt.

6.2 UBL Ordering–Prozess

Viele grundlegende Interoperabilitätsprobleme zwischen den Services wurden bereits durch das vorherige Minimalbeispiel entdeckt. Im Folgenden soll nun eine Integration mit bestehenden Services im größeren Stil durchgeführt werden. Details der Implementierung werden nur noch bei Neuerungen gegenüber dem vorherigen Fallbeispiel dargelegt. Ziel ist es dabei einen realistischen Anwendungsfall für die Validierung des Ansatzes heranzuziehen. Als Ausgangspunkt für die zu integrierenden Prozesse soll deshalb der Ordering–Prozess aus der *Universal Business Language (UBL)*–Spezifikation 2.1 [OAS11] dienen. Die UBL ist eine frei verfügbare Sammlung von standardisierten XML–basierten Geschäftsdokumenten, die unter dem Dach der OASIS entwickelt wird. Die weitverbreitete Nutzung, der XML, hat zu einer Vielzahl an unterschiedlichen, industriespezifischen Ausprägungen von universellen Dokumenten wie Bestellungen und Rechnungen geführt. Um die daraus entstehenden Nachteile in Bezug auf die Interoperabilität verschiedener Geschäftspartner zu überbrücken, wurde von der UBL ein generisches, erweiterbares XML–Austauschformat für Geschäftsdokumente definiert (vgl. [OAS11, S.8 f.]). Die UBL definiert neben den Dokumenttypen auch eine Reihe von Prozessen, in denen diese zum Einsatz kommen können. Abbildung 32 zeigt die Modellierung des Bestellprozesses.

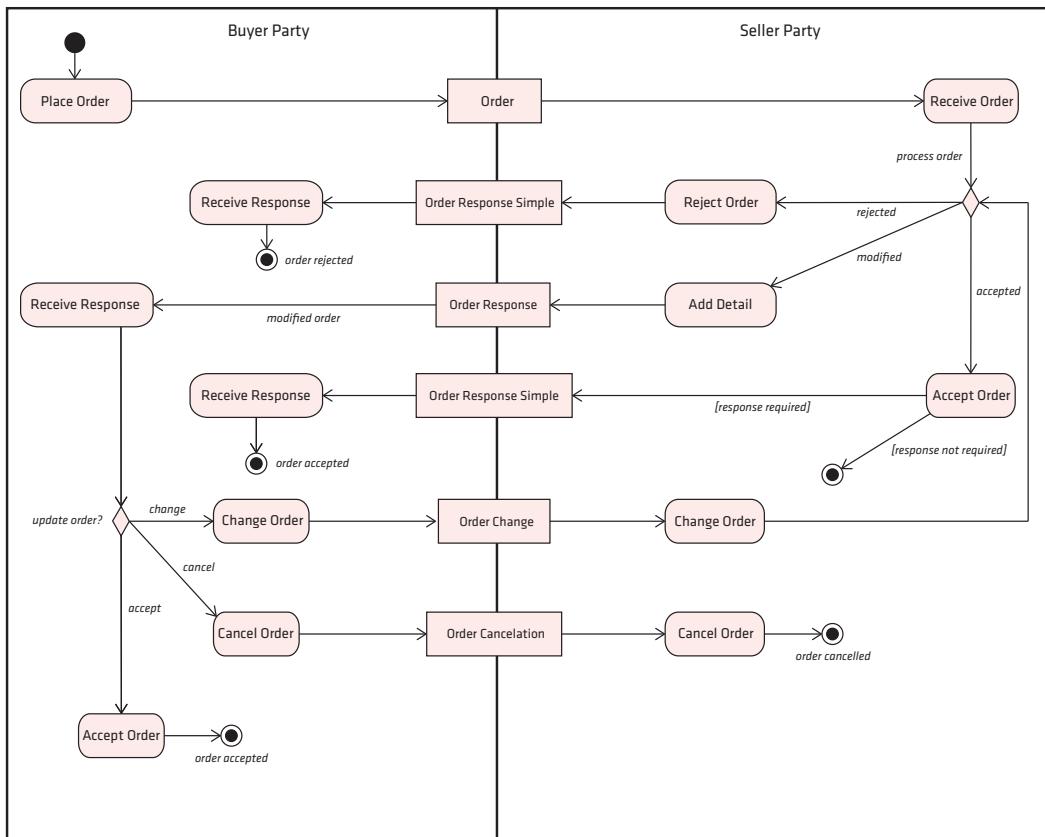


Abbildung 32: UBL Ordering–Prozess [OAS11, S.28]

In diesem Prozess interagieren zwei Parteien, ein Käufer und ein Verkäufer, miteinander, um eine Auftragserteilung durchzuführen. Es findet der Austausch verschiedener Dokumente statt, die durch die UBL spezifiziert sind. Die UBL definiert sowohl die Standarddatentypen, als auch die Nachrichtentypen für verschiedene Anwendungsfälle. Darüber hinaus bietet die UBL aber auch die Möglichkeit die Typen bei Bedarf zu erweitern (vgl. [OAS11, S.8]). Neben der erfolgreichen Auftragserteilung und einer Auftragsablehnung

umfasst der Prozess die Szenarien Bestellveränderung und Stornierung. Die Modellierung stellt hierbei keinen normativen Ablauf einer Bestellung dar, sondern soll vor allem den ausgetauschten Nachrichtenartefakten eine Semantik geben (vgl. [OAS11, S.12]). Die Prozesse können in der Praxis deshalb durchaus etwas anders verlaufen. Daraus folgt, dass unterschiedliche Implementierer zu unterschiedlichen Auffassungen kommen können, was zu Inkompatibilitäten führen kann. Deshalb scheint es legitim für das Fallbeispiel genau diese Situation nachzubilden und zwei konkrete Prozesse zu definieren, die zueinander zunächst inkompatibel sind und dies mit einem Adapter zu lösen. Betrachtet man den gegebenen Prozess genauer, fällt auf, dass der per se zunächst auf Kompatibilität ausgelegte Prozess bereits Inkompatibilitäten aufweist. Diese manifestieren sich sowohl im Kontrollfluss, als auch in der Beschreibung der Rahmenbedingungen. Es existieren beispielsweise nicht immer beidseitige Bestätigungen bei internen Entscheidungen, wie z.B. einer Modifikation einer Bestellung. Dies sind Kandidaten für Deadlocks im Kontrollfluss, da der Verkäufer nicht wissen kann, ob seine Bestellveränderung angenommen wurde oder nicht, v.a. weil die Situation in der Beschreibung als völlig neuer Zustand einer Bestelltransaktion deklariert wird. Nebenbedingungen wie die Möglichkeit eine Bestellung jeder Zeit stornieren zu können, während Verträge den Zeitpunkt fest legen an dem eine solche Stornierung ignoriert wird, sind dabei schon Deadlocks in der Modellierung, da für einen Partner bei einer internen Behandlung durch eine Partei nie klar ist, welche Entscheidung getroffen wurde (vgl. [OAS11, S.29]). Da solche Probleme jedoch auch nicht sinnvoll von einem Adapter gelöst werden können, müssen die Prozesse vorher entsprechend angepasst werden. Im Folgenden werden die beiden Ausgangsprozesse und eventuelle weitere Probleme vorgestellt und anschließend in eine Petri–Netz–Repräsentation transformiert. Danach wird ein Adapter generiert und implementiert, der die bestehenden Inkompatibilitäten zwischen den Services löst. Der Käufer wird dabei in BPEL implementiert, während der Verkäufer durch einen WF–Prozess abgebildet wird³⁰.

6.2.1 Seller Party

Abbildung 33 zeigt zunächst den WF–Prozess des Verkäufers. Er erwartet gemäß der Spezifikation zunächst eine Bestellung eines Kunden. Anschließend überprüft er die Verfügbarkeit der Waren, was hier durch eine Zufallsentscheidung abstrahiert wird. Je nach Ergebnis sendet er entweder ein `OrderResponseSimple`–Dokument, mit einer Ablehnung oder Bestätigung des gesamten Auftrags oder eine `OrderResponse` mit einem teilweise veränderten Auftrag. Im letzteren Fall wartet er anschließend auf eine Bestätigung oder eine Stornierung des Auftrages durch den Kunden. Als Artefakt für eine Bestätigung kommt der Dokumenttyp `OrderResponseSimple` zum Einsatz. Diese Operation wurde dem Ausgangsprozess hinzugefügt, um eine Situation zu verhindern, in der ein modifizierter Auftrag unbestätigt und somit in der Schwebe bleibt. Im Gegensatz dazu akzeptiert der konkrete Verkäufer keine käuferseitige Veränderung des Auftrages, wie in Abb.32 modelliert.

Die Hauptartefakte der UBL–Spezifikation stellen die standardisierten Datentypen für die Kommunikation zwischen den Geschäftspartnern dar. Diese sind per XSD spezifiziert. Im vorherigen Fallbeispiel wurden die Datentypen selbst definiert, realistischer ist es jedoch, dass diese Artefakte bereits definiert sind und verwendet werden sollen. Da die komple-

³⁰Alle Prozesse und Artefakte des Fallbeispiels befinden sich auf dem Datenträger im Ordner CASES\UBL-ORDERING.

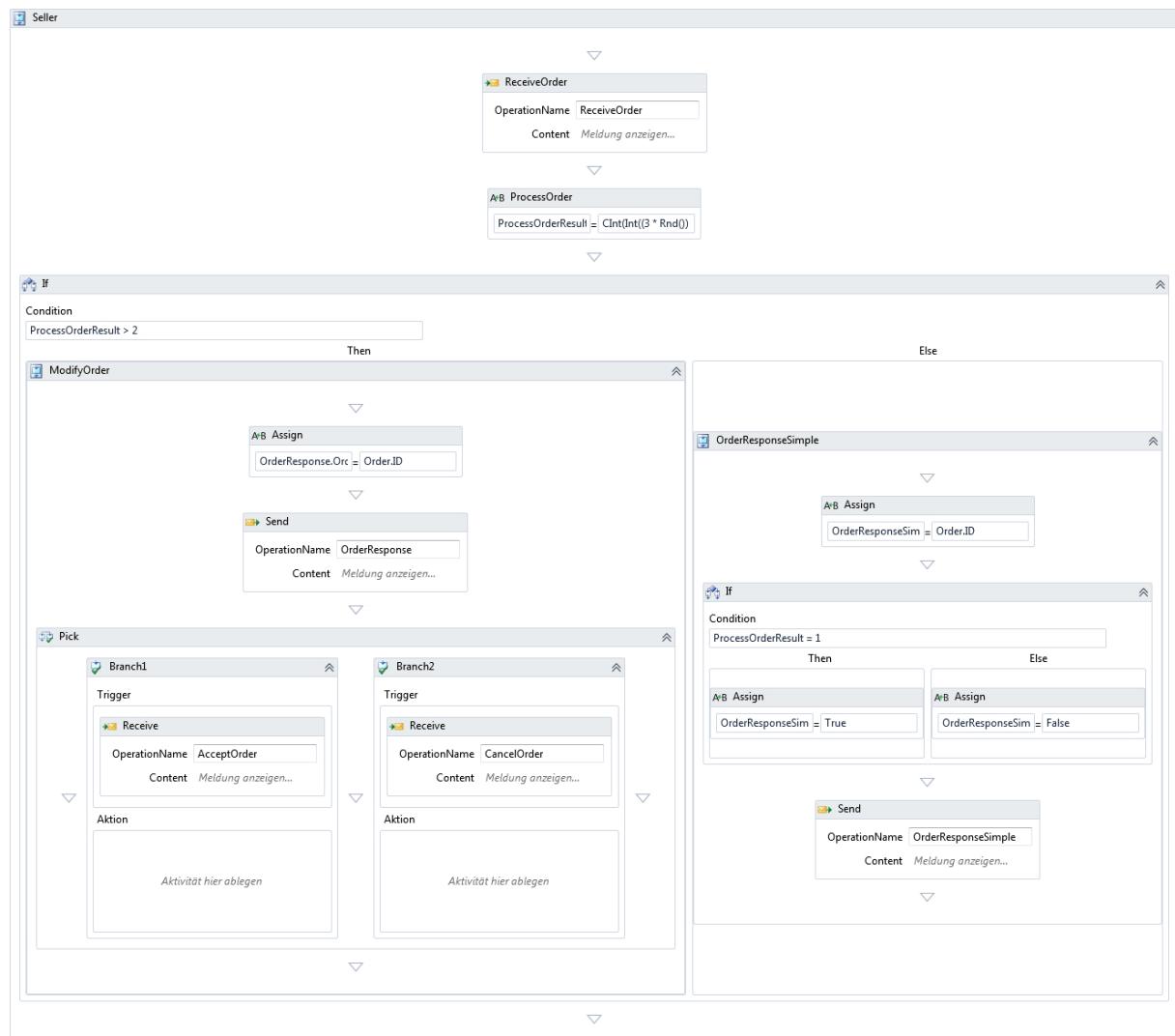


Abbildung 33: WF Ordering–Seller

ten Datentypen der UBL für das Fallbeispiel jedoch zu komplex und tiefgeschachtelt sind, wird eine der Komplexität angemessene Vereinfachung verwendet. Dabei sind die Dokumenttypen auf die geforderten Elemente beschränkt und diese jeweils als ein primitiver Typ definiert. Listing 16 zeigt hierzu beispielhaft den Dokumenttyp `Order`.

Listing 16: XSD des Dokumenttyps `Order`

```

1 <xsd:schema
2   xmlns="urn:oasis:names:specification:ubl:schema:xsd:Order-2"
3   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4   attributeFormDefault="unqualified"
5   elementFormDefault="qualified"
6   targetNamespace="
7     urn:oasis:names:specification:ubl:schema:xsd:Order-2"
8   version="2.0">
9
10  <xsd:element name="Order" type="OrderType"/>
11    <xsd:complexType name="OrderType">
12      <xsd:sequence>
13        <xsd:element maxOccurs="1" minOccurs="1" name="ID" type=
14          "xsd:int"/>

```

```

13      <xsd:element maxOccurs="1" minOccurs="1" name="IssueDate
14          " type="xsd:date"/>
15      <xsd:element maxOccurs="1" minOccurs="1" name="
16          BuyerCustomerParty" type="xsd:string"/>
17      <xsd:element maxOccurs="1" minOccurs="1" name="
18          SellerSupplierParty" type="xsd:string"/>
19      <xsd:element maxOccurs="unbounded" minOccurs="1" name="
20          OrderLine" type="xsd:string"/>
21      </xsd:sequence>
22  </xsd:complexType>
23</xsd:schema>

```

Die meisten Elemente stellen im Vergleich zur eigenen Definition in Kapitel 6.1.1 keine neuen Anforderungen an eine Umsetzung in WF als **DataContracts**. Wenn man jedoch das Element *OrderLine*, das für eine Auftragsposition steht, betrachtet, fällt auf, dass es mehrfach vorkommen kann. Durch die Deklaration würde dies bedeuten, dass mehrere gleich benannte Elemente *OrderLine* am Ende der Sequenz auftreten können, ähnlich einem Objekt–Array. Dieses Konzept deckt sich nicht mit denen gängiger objektorientierter Programmiersprachen, die immer ein umschließendes Objekt, z.B. eine Liste, die die Elemente enthält, fordern. Deshalb wird es von der Standardserialisierung in WF auch nicht unterstützt (vgl. [Mic12q]). Hierfür wäre ein zusätzliches Element nötig, welches eine beliebige Anzahl an *OrderLine*–Knoten enthalten könnte. Es wäre in diesem Fall diskutabel, ob man die Spezifikation nicht ändern sollte und solche Elemente äquivalent zur Darstellung in Listing 17 zu modellieren.

Listing 17: XSD einer objektorientierten Auftragsposition

```

1 <xsd:complexType name="OrderLine">
2   <xsd:sequence>
3     <xsd:element maxOccurs="unbounded" minOccurs="1" name="Line"
4         type="xsd:string"/>
5   </xsd:sequence>
6 </xsd:complexType>

```

Nichtsdestotrotz existiert die Spezifikation aktuell in der vorabgezeigten Variante und sollte daher so umgesetzt werden. Um in WF direkt mit XML–Daten arbeiten zu können, existiert die optionale Möglichkeit einer XML–Serialisierung. Die Serialisierung erlaubt es, auch solche Arrayfolgen zu codieren. Vom Prinzip funktioniert der **XmlSerializer** wie ein **DataContract**, durch Klassen mit entsprechenden Attributen. Arrayelemente müssen im Code mit dem **XmlElementAttribute** annotiert werden. Damit wird der **XmlSerializer** angewiesen das Array als eine Serie von XML–Elementen, anstatt als verschachtelte Menge von Elementen, zu serialisieren (vgl. [Mic12f]). Um die XSDs nicht manuell als Klassen definieren zu müssen, existiert das *XML Schema Definition–Tool*³¹. Der nachfolgende Aufruf generiert mit Attributen versehene C#–Klassen für alle in der Schemadatei enthaltenen Elemente:

```
xsd /c Money.xsd
```

Während der Implementierung stellt sich jedoch heraus, dass Probleme zwischen dem **XmlSerializer** und dem Visual Studio Designer bestehen. Versucht man wie gewohnt

³¹Das Tool **xsd.exe** ist standardmäßig im .NET Framework enthalten. Dokumentation unter [http://msdn.microsoft.com/de-de/library/x6c1kb0s\(v=vs.100\).aspx](http://msdn.microsoft.com/de-de/library/x6c1kb0s(v=vs.100).aspx)

eine Korrelation zu initialisieren und für das ausgewählte Handle ein entsprechendes Attribut per XPath auswählen, so ist dies nicht möglich, da das Auswahlfeld deaktiviert ist. Selektiert man in einer weiteren korrelierten `Receive`-Aktivität nun das Handle und wählt für `CorrelatesOn` ein entsprechendes Attribut aus, so funktioniert dies. Durch einem Blick auf den Xaml-Code (siehe Listing 18) lässt sich jedoch schnell feststellen, dass implizit der `DataContractSerializer` verwendet wurde und die Queries dementsprechend falsch sind. Hier wird beispielsweise die Eigenschaft `ID` über einen Feldnamen addressiert, der in der XSD-Repräsentation nicht existiert. Eine Verwendung von inhaltsbasierter Korrelation ist somit über den üblichen Weg nicht möglich. Über eine manuelle Codeanpassung kann die Korrelation jedoch entsprechend funktionsfähig eingerichtet werden (siehe Listing 19). Die Tests zeigen, dass die dahintersteckende Technik auch zusammen mit dem `XmlSerializer` funktioniert, der Designer jedoch einen Bug aufweist³².

Listing 18: XML-Korrelation Bug

```

1 <Receive.CorrelatesOn>
2   <XPathMessageQuery x:Key="key1">
3     <XPathMessageQuery.Namespaces>
4       <ssx:XPathMessageContextMarkup>
5         <x:String x:Key="xg0">
6           http://schemas.datacontract.org/2004/07/
7         </x:String>
8       </ssx:XPathMessageContextMarkup>
9     </XPathMessageQuery.Namespaces>
10    sm:body()/xg0:OrderType/xg0:idField
11  </XPathMessageQuery>
12 </Receive.CorrelatesOn>
```

Listing 19: XML-Korrelation Fix

```

1 <Receive.CorrelatesOn>
2   <XPathMessageQuery x:Key="key1">
3     <XPathMessageQuery.Namespaces>
4       <ssx:XPathMessageContextMarkup>
5         <x:String x:Key="xg0">
6           urn:oasis:names:specification:ubl:schema:xsd:OrderCancellation-2
7         </x:String>
8       </ssx:XPathMessageContextMarkup>
9     </XPathMessageQuery.Namespaces>
10    sm:body()/xg0:Order/xg0:ID
11  </XPathMessageQuery>
12 </Receive.CorrelatesOn>
```

Bestehen die vorangegangenen Einschränkungen im Schema nicht, können auch Klassen für den Standardserialisierer `DataContractSerializer` generiert werden. Hierfür existiert das *ServiceModel Metadata Utility*³³. Im konkreten Fall wurde wegen der Korrelationsprobleme `OrderLine` in ein einfaches Feld geändert. Der nachfolgende Aufruf generiert passende C#-Klassen mit `DataContract`-Attributen aus einer XSD:

svcutil /dconly Money.xsd

³²Die Ergebnisse wurden an Microsoft zur Begutachtung weitergeleitet (siehe <http://social.msdn.microsoft.com/Forums/en-US/wfprerelease/thread/46161181-2b6c-4bc9-bea3-011fd6b11fc9>)

³³Das Tool `svcutil.exe` ist standardmäßig im .NET Framework enthalten. Dokumentation unter <http://msdn.microsoft.com/de-de/library/aa347733.aspx>

Nach Behebung dieser Probleme kann der Prozess in ein Petri–Netz transformiert werden. Zur Transformation des WF–Prozesses kommt der in Kapitel 5 vorgestellte Prototyp zum Einsatz. Ebenso wie die bereits verwendeten Tools kann er über die Kommandozeile ausgeführt werden. Durch Übergabe der zu transformierenden Xaml(x)–Datei und einem optionalen Parameter */o* für die Ausgabedatei, erzeugt er ein Petri–Netz des Prozesses.

wf2owfn /*i*=Seller.xamlx /*o*=Seller.owfn

Abbildung 34 zeigt das resultierende Netz. Vergleicht man die Abbildung mit dem Ausgangsprozess in Abb.33 kann man erkennen, wie die einzelnen Konstrukte umgesetzt wurden. Nach dem Empfang des Auftrages spaltet sich der Kontrollfluss zunächst in die beiden Zweige der If–Aktivität. Die rechte Verzweigung bestätigt nach einer weiteren If–Entscheidung und einer entsprechenden Zuweisung die Ablehnung oder Annahme der Bestellung mit einer Nachricht. Auch im linken Ast spaltet sich der Kontrollfluss nach dem Senden der Änderungsnachricht erneut. Hier kann man die Pick–Aktivität erkennen, die je nach erhaltenener Nachricht den Prozess mit der Ablehnung oder der erfolgreichen Abwicklung der Bestellung abschließt.

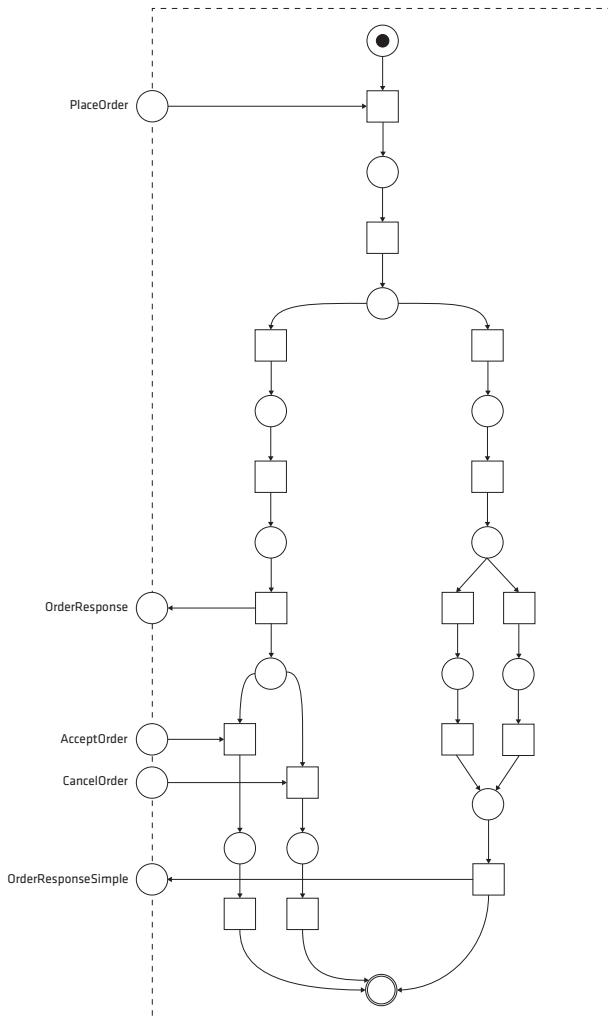


Abbildung 34: oWFN Ordering—Seller

6.2.2 Buyer Party

Abbildung 35 zeigt nun den ausführbaren BPEL-Prozess des Käufers. Er implementiert alle möglichen Abläufe aus dem Ausgangsmodell. Die Aktivitäten *ReceiveInput* und *ReplyResult* dienen erneut lediglich zum Starten und Überwachen des Ergebnisses der Orchestration. Zunächst erteilt der Prozess einen Auftrag und erwartet solange (*RepeatUntil*-Aktivität) einen der beiden definierten Nachrichtentypen als Antwort, bis der Auftrag abgelehnt oder akzeptiert wird. Im Falle einer *OrderResponseSimple* wird die Bestellung entweder komplett bestätigt oder abgelehnt. Erhält der Prozess eine *OrderResponse*, so wurden Teile des initialen Auftrags modifiziert. Intern entscheidet der Käufer sich entweder die Modifikation anzunehmen, abzulehnen oder nochmals zu ändern. Dies wird im Prozess durch eine „zufällige“ Entscheidung mit einer Modulo-Operation auf der Nutzeridentifikation abstrahiert, da XPath 1.0 keine Zufallszahl liefern kann. Bei allen drei Entscheidungen schickt der Prozess entsprechende Dokumente zum Verkäufer. Zusätzlich zu den in der UBL spezifizierten Dokumenten wird auch bei einer Annahme der veränderten Bestellung eine Bestätigung versendet, um einen nicht lösbar Deadlock zu vermeiden. Hierfür wird der Dokumententyp *OrderResponseSimple* wiederverwendet.

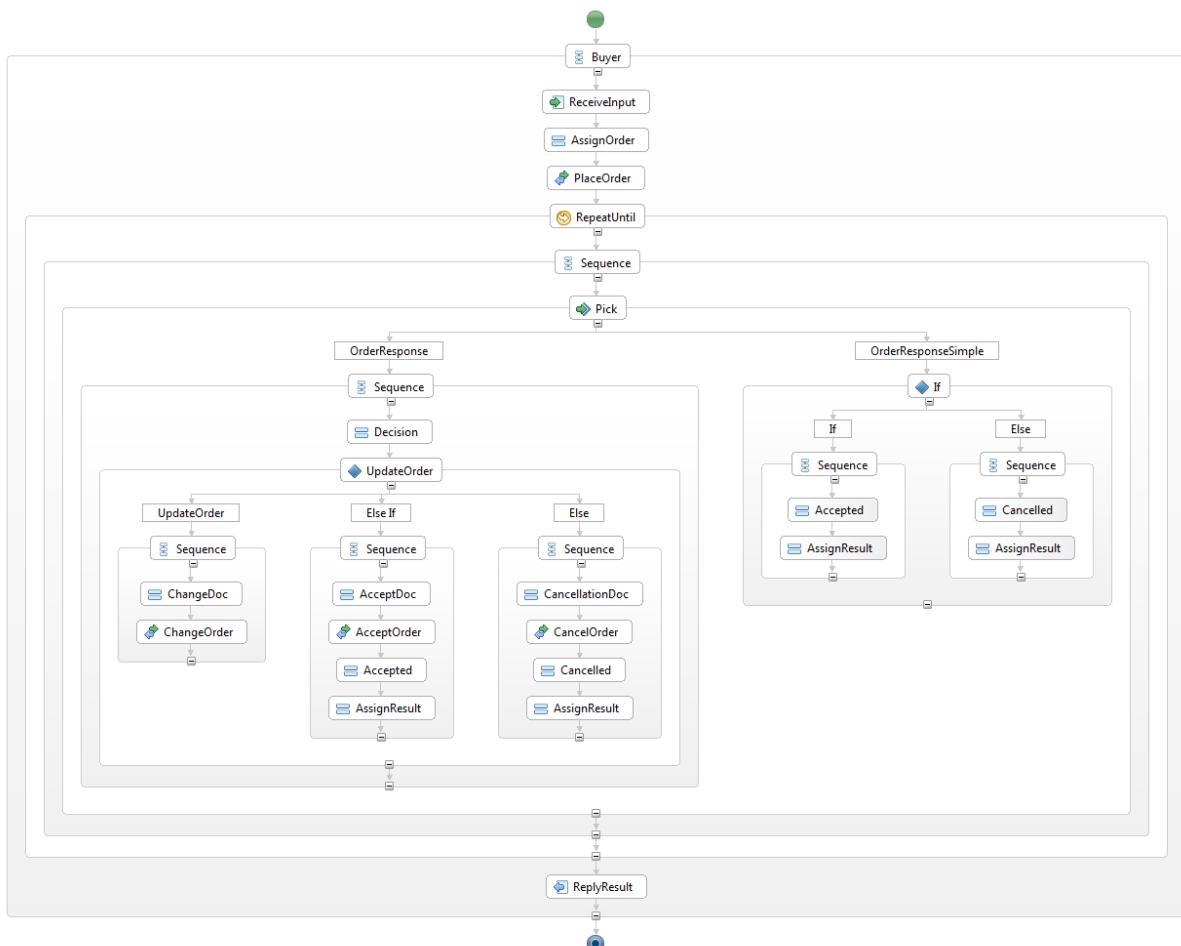


Abbildung 35: BPEL Ordering–Buyer

Bei der Implementierung entstehen keine weiteren neuen Probleme mehr, weswegen direkt auf die Transformation des Prozesses in ein Petri–Netz eingegangen werden soll. Mit dem Tool *BPEL2oWFN* soll eine entsprechende Transformation durchgeführt werden.

Das Tool funktioniert vom Prinzip her ähnlich wie der WF2oWFN–Prototyp, besitzt jedoch einen höheren Funktionsumfang und eine größere Produktreife. Mit dem folgenden Aufruf wird ein Petri–Netz ohne Fehler–, Compensation–, oder Termination–Behandlung (*-p small*) erstellt, passend zur Ausgabe des Prototypen:

```
bpel2owfn -m petrinet -p small -f oPFN --output=Buyer.oPFN Buyer.bpel
```

Beim Versuch Quellcode zu transformieren, der teilweise mit dem BPEL–Designer erstellt wurde, kommt es neben vielen übersprungenen Syntaxfehlern mehrfach zum Abbruch der Transformation. Bei der Initialisierung von Literalen lassen sich die Syntaxfehler trotz nach der BPEL–Spezifikation syntaktisch korrektem Code nicht beseitigen. Auch werden gültige XPath–Ausdrücke als Fehler gemeldet. Dazu gehören auch Probleme mit *Character Data (CDATA)*–Deklarationen³⁴, die der Designer um alle XPath–Ausdrücke hüllt. Bei den bisher genannten Problemen handelt es sich jedoch nur um Warnungen, die eine Transformation zulassen. Einige Prozesse lassen sich jedoch aufgrund von Syntaxfehlern gar nicht transformieren. Die betroffenen Prozesse laufen jedoch problemlos in Apache ODE. Grund dafür sind Probleme mit Reihenfolgebeziehungen innerhalb des BPEL–Codes, die zur Nichtkompilierung führen. BPEL2oWFN hält sich dabei normativ an die definierten Reihenfolgen innerhalb der BPEL–Spezifikation, während der BPEL–Designer, als auch Apache ODE in diesem Hinblick vergleichsweise tolerant sind. Listing 20 zeigt den Code einer *pick*–Aktivität, die mit dem BPEL–Designer erstellt wurde. Innerhalb des *onMessage*–Knotens folgt zuerst die durchzuführende Aktivität und dann die Korrelation. Korrekt im Sinne der BPEL–Spezifikation ist jedoch die Reihenfolge in Listing 21, die zunächst die Korellation und anschließend die Aktivität beherbergt. Hier sollte im BPEL–Designer nachgebessert werden. Sind alle Fehler beseitigt, liefert BPEL2oWFN das Netz in Abb.36a³⁵.

Listing 20: pick BPEL–Designer

```

1 <pick>
2   <onMessage partnerLink="SellerService" operation="OrderResponse">
3     <sequence name="DoWork"/>
4     <correlations>
5       <correlation set="SharedHandle" initiate="no"/>
6     </correlations>
7   </onMessage>
8 </pick>
```

Listing 21: pick BPEL–Spezifikation

```

1 <pick>
2   <onMessage partnerLink="SellerService" operation="OrderResponse">
3     <correlations>
4       <correlation set="SharedHandle" initiate="no"/>
5     </correlations>
6     <sequence name="DoWork"/>
7   </onMessage>
8 </pick>
```

³⁴Mit einem CDATA–Abschnitt wird einem XML–Parser mitgeteilt, dass kein Markup folgt, sondern normaler Text der nicht geparsed werden soll (vgl. [W3C08]).

³⁵Für eine verbesserte Anschaulichkeit der folgenden Betrachtungen ist das Netz zudem manuell reduziert worden.

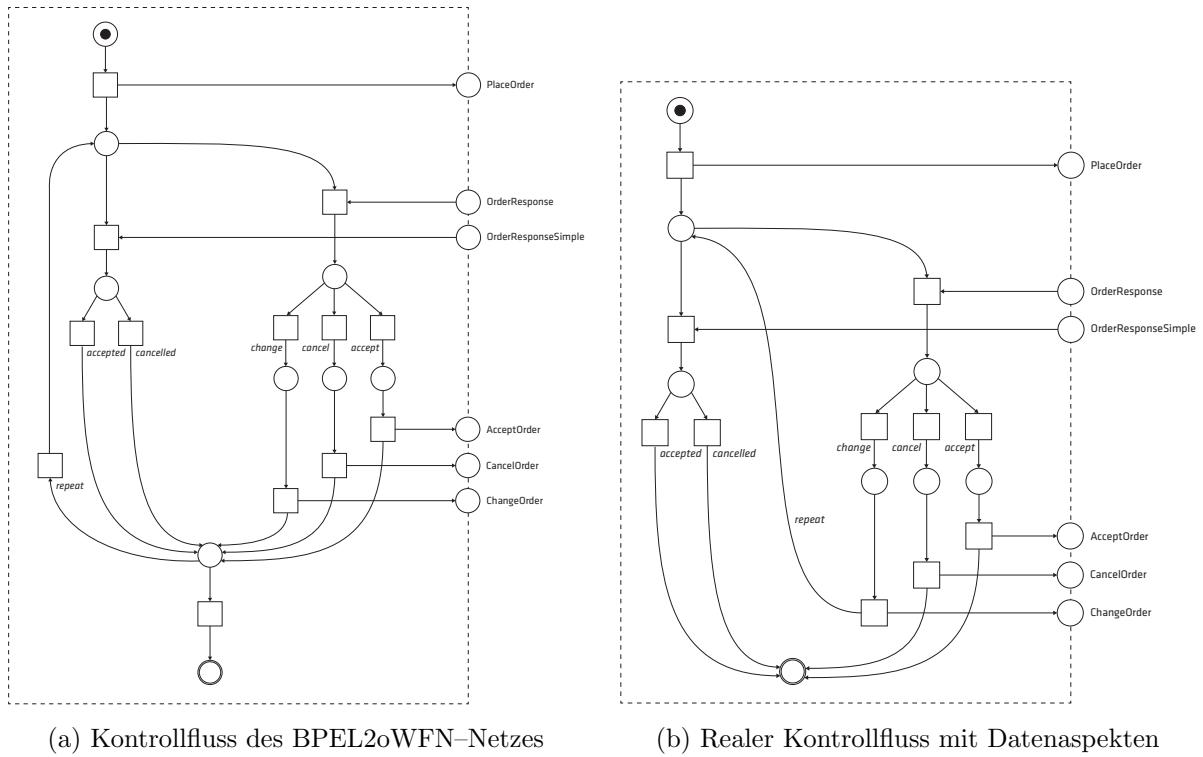


Abbildung 36: oWFN-Netze Buyer

6.2.3 Adaptersynthese und Integration

Betrachtet man nur die beiden Ausgangsprozesse, so fällt auf, dass beide unverändert noch nicht miteinander kompatibel sein können. Der Käufer unterstützt die Möglichkeit eine eigene Änderung an einem modifizierten Auftrag des Verkäufers vorzunehmen, während der Verkäufer diese Möglichkeit nicht anbietet. Alle anderen Operationen werden von beiden Partnern unterstützt. Wie es scheint, muss lediglich diese Inkompatibilität bereinigt werden, um die Komposition kompatibel zu machen. Wie bekannt ist, muss um einen Adapter zu generieren eine Definition aller erlaubten Nachrichtentransformationen angegeben werden. Da noch keine automatische Ableitung dieser SEA möglich ist, muss sie manuell definiert werden. Dies kann durch eine Betrachtung der bestehenden Probleme und einer Ableitung der nötigen Regeln um diese zu lösen, geschehen.

Da die Bezeichnungen der Input- und Outputstellen der beiden Netze auch bei syntaktisch gleichen Nachrichtentypen nicht äquivalent sind, müssen auch hier für eine virtuelle Transformation entsprechende Regeln bereitgestellt werden. Die Regeln eins bis fünf in Tabelle 2 sorgen für eine korrekte Weitergabe der Nachrichten vom einen zum anderen Partner. Wie vorhin angesprochen, unterstützt der Verkäufer keine erneute Veränderung des Auftrags als Antwort auf eine von ihm modifizierte Bestellung, sondern lediglich eine Bestätigung oder Stornierung. Damit der Prozess im Sinne des Käufers regelgerecht ablaufen kann, könnte der Käufer beispielsweise durch eine vom Adapter selbstgenerierte Ablehnung seiner Änderungsanfrage dazu gebracht werden diese zu akzeptieren. Da der Verkäufer zudem eine Ablehnung oder Stornierung erwartet, müsste der Adapter in der Lage sein, ebenso eine Ablehnung zu generieren. Damit würde das Problem gelöst wer-

den und der Prozess bei beiden Partnern mit einer Stornierung enden. Regel sechs zeigt die entsprechende Umsetzung, bei der die Änderungsanfrage (`ChangeOrder`) in die beiden Nachrichten für eine Stornierung des Käufers (`CancelOrder`) und eine des Verkäufers (`OrderResponseSimple`) aufgespalten wird.

Tabelle 2: SEA Ordering–Adapter

Regel 1:	s.OrderResponseSimple	→	b.OrderResponseSimple;
Regel 2:	s.OrderResponse	→	b.OrderResponse;
Regel 3:	b.PlaceOrder	→	s.ReceiveOrder;
Regel 4:	b.AcceptOrder	→	s.AcceptOrder;
Regel 5:	b.CancelOrder	→	s.CancelOrder;
Regel 6:	b.ChangeOrder	→	s.CancelOrder, b.OrderResponseSimple;

Mit dem bekannten Aufruf des Tools Marlene lässt sich jedoch kein Adapter generieren.

```
marlene buyer.owfn seller.owfn -t arbitrary -o adapter.owfn -r rulefile.ar
```

Zur Identifikation des Problems empfiehlt es sich erneut das entstandene Petri–Netz in Abb.36a zu betrachten. Je nach Typ der empfangenen Entscheidung begibt sich der Kontrollfluss in einen der beiden Zweige. Anschließend spaltet er sich erneut auf und symbolisiert durch die gesendete Nachricht, welche Entscheidung getroffen wurde. Schlussendlich laufen alle Zweige wieder in eine Stelle zusammen. Genau hier kommt es zu einem Problem. Der Kontrollfluss wird nun entweder zurück zum Anfang der *RepeatUntil*–Aktivität geführt oder der Prozess terminiert. Diese Entscheidung wird im BPEL–Prozess anhand der Auswertung der Variablen getätigt. Im Petri–Netz jedoch ist diese Entscheidung nicht deterministisch. Dies führt zu einer internen Entscheidung, die einem Partnernetz nicht zugänglich ist und zu einem Deadlock führen kann. Tatsächlich kann gleichwohl, bezieht man den Datenaspekt mit ein, nur bei der Ausführung des *ChangeOrder*–Falls der Kontrollfluss erneut den Rücksprung vollführen, da lediglich im Falle einer Veränderung des Auftrages der auszuwertende Ausdruck einen wahren Wert ergibt. Das Petri–Netz bildet dies jedoch nicht so ab und lässt bei vier von fünf Möglichkeiten fälschlicherweise eine Rückführung zu. Dies führt zu dem Problem, dass der Käufer u.U. nochmals eine Nachricht erwartet. Auch nach den Regeln des Adapters ist es nicht möglich diesem Bedürfnis zu entsprechen. Der Adapter kann selbst keine solche Nachricht mehr generieren und auch vom Verkäufer nicht mehr erhalten, da dieser bereits terminiert ist. Dadurch entsteht ein Deadlock und das Gesamtnetz ist nicht steuerbar. Hier zeigt sich ein generelles Problem bei der Umsetzung von datenbasierten Konstrukten der Workflow–Sprachen in Petri–Netze. Durch die freie Wahlmöglichkeit und die Abstinenz des Datenaspektes können ungewollte Auswirkungen auf den Kontrollfluss des Netzes entstehen und somit einen von der Programmierung her steuerbaren Service nicht steuerbar werden lassen. Dies betrifft vor allem Konstrukte, die ihre Entscheidungen anhand von Daten treffen, wie z.B. bedingten Anweisungen oder Schleifen. Dieses Problem lässt sich zudem auch auf die Abstinenz des Zeitaspektes ausweiten. Als einfaches Beispiel kann eine *pick*–Aktivität dienen, die eine Nachricht erwartet oder nach einer gewissen Zeit den Kontrollfluss erneut freigibt. Existieren hier keine expliziten Nachrichten, die den Weg des Netzes erkennen lassen, entsteht das gleiche Problem wie bei der Abstinenz des Datenaspektes. Dies wirft die Frage auf in wieweit eine praktisch anwendbare Integration ohne Daten– und Zeitaspekte tatsächlich sinnvoll ist. Gerade da reale Prozesse im Allgemeinen komplexe Strukturen haben, ist es schwer möglich solche fehlerhaften Strukturen manuell aufzuspüren.

Das angesprochene Problem lässt sich im konkreten Fall wie folgt lösen: Mit Kenntnis des Datenaspektes kann das Netz z.B. wie in Abb.36b manuell angepasst werden. Lediglich der *ChangeOrder*-Zweig macht nun einen Rücksprung zum Beginn der *RepeatUntil*-Schleife, da bei einer Veränderung des Auftrags eine erneute Bestätigung erwartet wird. Dies simuliert das Datenverhalten, welches abstrahiert wurde. Der Kontrollfluss bleibt dabei weiterhin in der Weise erhalten, wie er mit Datenaspekten implementiert wurde. Offensichtlich ist das resultierende Netz nun auch so steuerbar wie es intendiert ist. Es existieren keine internen Entscheidungen mehr, die für einen Partner unentdeckt bleiben können. Dies ermöglicht nun auch die Generierung eines Adapters für die beiden Services. Abbildung 37 zeigt den generierten Adapter.

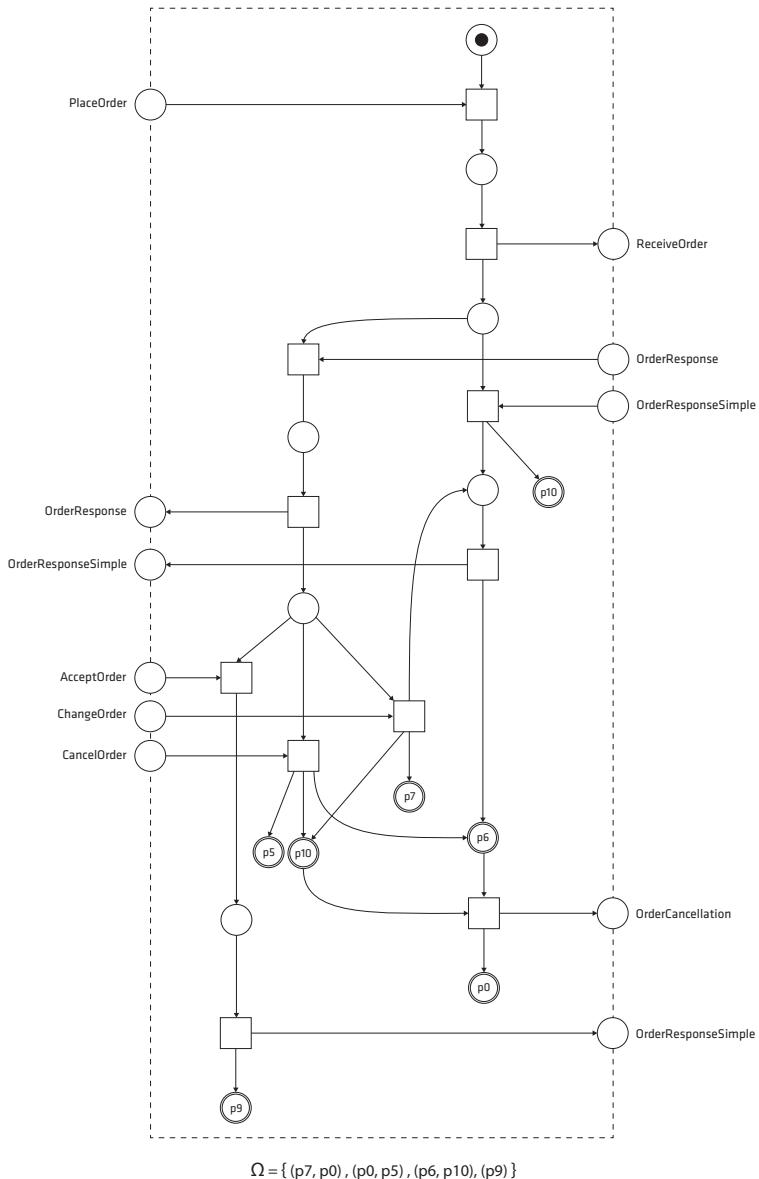


Abbildung 37: oWFN Ordering-Adapter

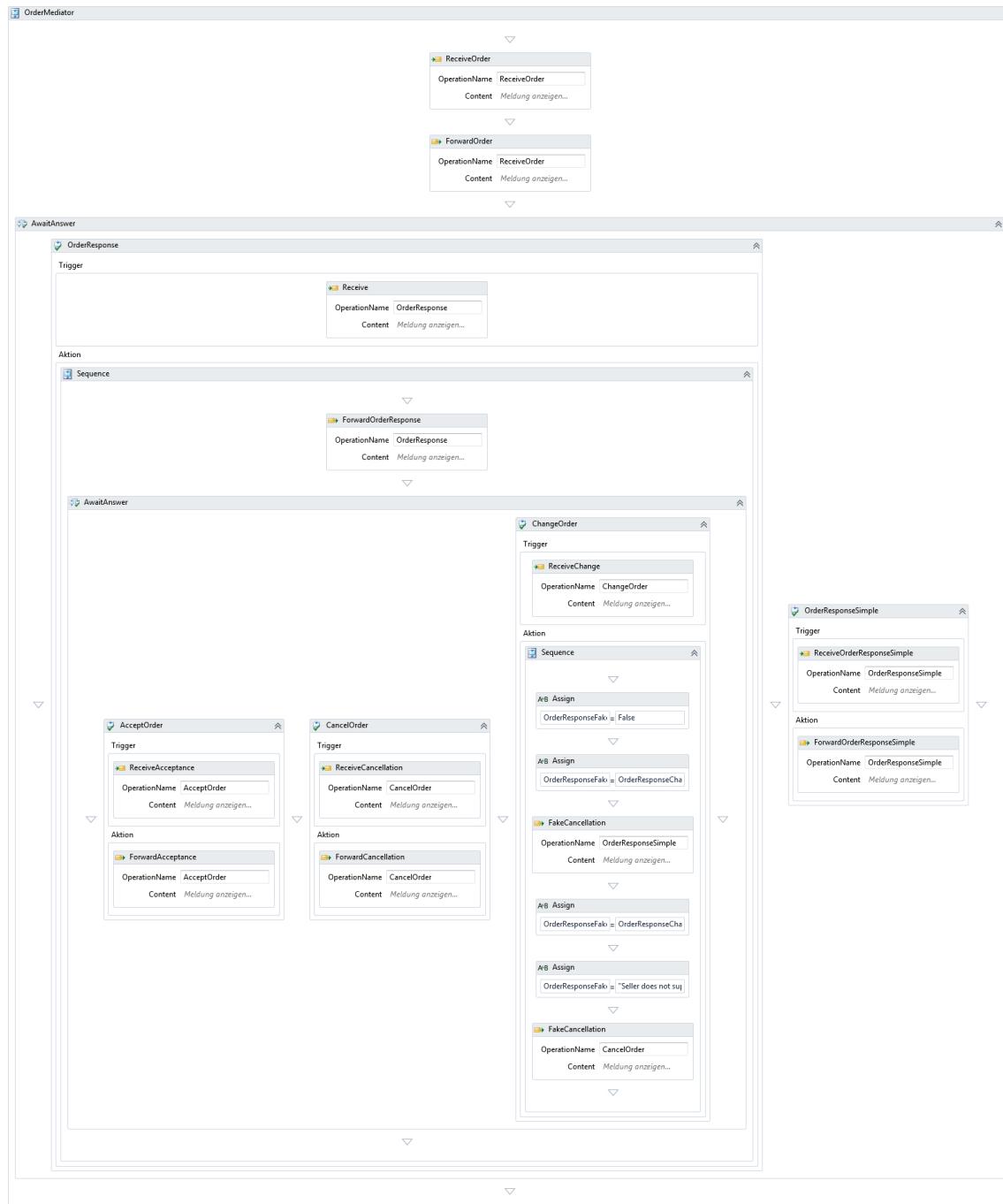


Abbildung 38: WF Ordering-Mediator

Wie in Kapitel 6.1.3 angesprochen, gibt es große Probleme mit dem Tool oWFN2BPEL bei Übersetzung der Petri-Netze in abstrakte BPEL-Prozesse. Da der vorherige Adapter schon in BPEL implementiert wurde, soll das Petri-Netz des Adapters dieses Mal manuell in einen entsprechenden WF-Prozess transformiert werden. Hierfür wird ein möglicher Weg durch den Adapter vollzogen und durch entsprechende Konstrukte in WF modelliert. Nach dem Empfang der Bestellung des Käufers wird diese direkt an den Verkäufer weitergeleitet. Anschließend wird auf die Antwort des Verkäufers gewartet. Dies wird durch eine Pick-Aktivität implementiert, die auf eine der beiden Nachrichten `OrderResponse` oder `OrderResponseSimple` wartet. Wird letztere empfangen, wird diese an den Käufer weitergeleitet und der Prozess befindet sich in einem finalen Zustand (*p10, p6*). Erhält der Adapter stattdessen eine `OrderResponse`, also eine Modifikation des Auftrages, wird

diese weitergeleitet und danach wieder mit einer `Pick`-Aktivität eine der drei möglichen Antworten erwartet. Bei `AcceptOrder` und `CancelOrder` wird die Nachricht direkt an den Verkäufer transferiert und der Prozess ebenfalls beendet ($p9$ und $(p0, p5)$). Im Falle einer `ChangeOrder` wird an den Verkäufer eine Stornierung und an den Verkäufer eine Ablehnung seiner Anfrage ($p7, p0$) gesendet und das Fallbeispiel damit erfolgreich gelöst. Die Abbildung 38 zeigt den implementierten WF-Prozess in der Gesamtheit.

7 Future Work

Im Laufe der Arbeit wurden bereits viele Pattern der Standardaktivitäten definiert und implementiert, dennoch ist noch nicht die vollständige Semantik für WF abgebildet. Neben dem Ziel möglichst viele weitere Aktivitäten zu integrieren, gilt es vor allem die Netze um den kompletten Kontrollfluss zu erweitern. Die Pattern müssen dazu mit den Aspekten der Fehlerbehandlung und der Abbruchbehandlung angereichert werden. Im Zuge dessen ist es auch möglich, die bei einigen Pattern getroffenen Einschränkungen (z.B. fehlende `CompletionCondition` bei `Parallel`) zu korrigieren.

Im März 2012 wurde die Basisspezifikation der Xaml [Mic12e] von Microsoft veröffentlicht. Darunter auch die Vokabulare für die WPF und Silverlight. Die in Anhang A aufgeführte Spezifikation wurde erarbeitet, als noch keinerlei Spezifikationen vorhanden waren. Durch die Veröffentlichung der Xaml–Basisspezifikation ergeben sich neue Möglichkeiten, da sie genaueren Aufschluss über den Aufbau der abgeleiteten Vokabulare gibt. Auch wenn noch keine spezielle Spezifikation für WF veröffentlicht ist, kann mit der Xaml–Spezifikation und der Dokumentation der WF–Klassen [Mic12m, Mic12n] die Spezifikation in Anhang A größtenteils verifiziert und ggf. vervollständigt werden. Zudem könnte die Spezifikation mit geringerem Aufwand mit den noch fehlenden Aktivitäten komplettiert werden. Dies würde es möglich machen den Compiler mit einer statischen Analyse anzureichern, die die Validität der WF–Prozesse prüfen kann. Aktuell sind diese Tests auf die Wohlgeformtheit der Xaml–Struktur und die in den Modulen implementierten Syntaxprüfungen beschränkt. Mit Kenntnis der kompletten Spezifikation wäre es dann auch möglich, Transformationen von anderen nicht ausführbaren oder ausführbaren Sprachen (z.B. BPMN, BPEL) nach WF durchzuführen. Dies scheint gerade wegen der Produktreife der Workflow Foundation ein interessanter Ansatz.

Die in dem API des Prototypen vorhandenen Petri–Netz–Klassen beinhalten aktuell nur die vom Compiler benötigte Funktionalität. Die Klassensammlung könnte zu einer vollständigen, unabhängig nutzbaren Petri–Netz–API für .NET ausgebaut werden. Zum jetzigen Zeitpunkt konnte noch kein solches API für das .NET Framework identifiziert werden. Da Petri–Netze in vielen Anwendungsbereichen nützlich sein können (vgl. [RR98, Van98, HR04, Mur89]), könnte ein .NET–basiertes API hilfreich sein und müsste ggf. nicht wie in diesem Falle neu implementiert werden.

Während der Bearbeitung der Fallbeispiele sind einige Probleme in der Interoperabilität der Systeme aufgefallen. Um die Integration zwischen heterogenen Prozesssystemen zu ermöglichen, müssen neben der Kontrollflussintegration auch die Aspekte der Kommunikation zwischen den Systemen weiter beleuchtet werden. Selbst zwischen Webservices, die eigentlich standardisiert sind, herrschen deutliche Unterschiede in der Implementierung und Abdeckung der Funktionalität (vgl. [SV09]). Auch auf diesem Gebiet fehlen entsprechend umfangreiche Evaluationen. Hier gilt es separat weitere Bestrebungen zu verfolgen, um die Interoperabilität der Systeme zu verbessern.

8 Fazit

Ziel der Arbeit war es, eine automatisierte Integration zwischen heterogenen prozessbasierten Systemen in Service-orientierten Architekturen mit Hilfe von Petri-Netzen durchzuführen. Hierzu wurden Laufzeitsysteme mit den beiden Workflowsprachen BPEL und WF ausgewählt, die über Web Services miteinander interagieren sollten. Im Fall von BPEL wurde auf die Ergebnisse und Tools aus [Loh07] zurückgegriffen. Für WF wurde für eine ausgewählte Untermenge der Aktivitäten eine Petri-Netz-Semantik erstellt, welche im modular erweiterbaren Prototypen WF2oWFN implementiert ist, um eine automatisierte Transformation von WF-Prozessen in oWFN zu ermöglichen. In diesem Zusammenhang wurde zudem für WF eine Spezifikation für die implementierten Aktivitäten erarbeitet, die die XML-basierte Darstellung der Prozesse definiert. Anhand zweier Fallbeispiele konnte die Machbarkeit der Konzepte nachgewiesen werden. Mithilfe eines realitätsnahen Beispiels eines Bestellprozesses aus der UBL [OAS11], wurde zudem erstmals eine automatisierte Integration zwischen zwei heterogenen Workflowsprachen mit Petri-Netzen durchgeführt. Dabei wurde die komplette Prozesskette von der Transformation der Prozesse in Petri-Netze über die Synthese eines Adapters und dessen Transformation zurück in eine ausführbare Workflowbeschreibung dargelegt und erfolgreich durchgeführt. Gerade die Abstinenz des Datenaspektes zeigte bei dem Realbeispiel jedoch Einschränkungen in der Anwendbarkeit der Konzepte auf, da steuerbare Prozesse auf Petri-Netz-Ebene ohne Anpassungen nicht mehr steuerbar waren. Für eine praxistaugliche Anwendbarkeit müssen diese Aspekte in die Analyse mit einbezogen werden. Neben dem Kontrollfluss konnten zudem auch Probleme in der Kommunikation der Prozesssysteme über Webservices aufgedeckt werden. Viele der von den Software-Anbietern implementierten Standards sind häufig nur teilweise oder unterschiedlich interpretiert umgesetzt. Hier gilt es weiter die Bemühungen voran zu treiben, um eine wirkliche Interoperabilität in der Webservice-Kommunikation zu ermöglichen.

Neben den Hauptzielen konnten weitere Ergebnisse in verbundenen Bereichen entdeckt werden. Durch die Implementierungen der Fallbeispiele und deren Ausführung in den Workflow Engines konnten Informationen über die Reife und bestehende Probleme der Produkte gewonnen werden. WF besitzt zusammen mit dem Visual Studio ein stabiles, zuverlässiges System, während BPEL mit Apache ODE und dem Eclipse-BPEL-Designer noch keine produktive Arbeitsweise erlaubt. Zum einen weist Apache ODE noch keine komplette Unterstützung der BPEL-Spezifikation auf, zum anderen können die Vorteile der graphischen und intuitiven Modellierung von Prozessen mit dem Eclipse-BPEL-Designer wegen vieler Bugs quasi nicht genutzt werden. Eine manuelle Modellierung der BPEL-Prozesse bleibt für den produktiven Einsatz mehr als fraglich. Hier empfiehlt es sich ebenso, kommerzielle Produkte zu Rate zu ziehen.

Die Arbeit hat gezeigt, dass die Herausforderungen bei der Integration von prozessbasierten Systemen in einigen Fällen bereits erfolgreich gelöst werden können und fundierte Methodiken zur Bewältigung dieser zur Verfügung stehen. Es hat sich jedoch auch herausgestellt, dass noch viele offene Fragen und Probleme bestehen, die in zukünftigen Forschungen bewältigt werden müssen.

A Xaml(x)–Spezifikation

Konventionen der Notation

Zur Definition der Grammatik von Xaml(x)–Dateien wird auf eine informale Syntax zurückgegriffen, wie sie auch in der BPEL–Spezifikation [OAS07] verwendet wird. Die Syntax wird wie eine XML–Instanz dargestellt, jedoch stellen die Werte Datentypen statt konkrete Daten dar. Ausdrücke in Großbuchstaben und dunkelblauer Farbe sind Referenzen auf Basisdatentypen (z.B. **QNAME**). Schwarze Betonungen stellen Referenzen auf eine ausgelagerte Definition dar (z.B. **standard-attributes**). Attribute sind in Lila gekennzeichnet (z.B. **DisplayName**), Elemente in Türkis (z.B. **WriteLine**). Darüber hinaus können Elementen und Attributen Quantoren angefügt werden. Existiert kein Quantor, so muss das Element genau einmal vorkommen. „?“ steht dabei für ein einmaliges optionales (0 oder 1), „*“ für ein optionales mehrmaliges Vorkommen (0 oder mehr). Elemente oder Attribute mit dem Quantor „+“ müssen mindestens einmal existieren (1 oder mehr). Die Zeichen „[“ und „]“ gruppieren die beinhalteten Elemente. Elemente und Attribute, die mit „|“ getrennt sind und durch die Klammern „(“ und „)“ gruppiert sind, stellen syntaktische Alternativen dar. Die Reihenfolge der Elemente und Attribute auf einer Gültigkeitsebene ist dabei nicht normativ, d.h. sie können u.U. auch in anderer Anordnung auftreten. Die dargestellten Fragmente sind für sich alleine nicht lauffähig. Sie müssen sich innerhalb eines **Activity**–Wurzelementes (bei Xaml–Dateien) oder eines **WorkflowService**–Wurzelementes (bei Xamlx–Dateien) befinden, in dem die referenzierten Namespaces definiert sind. Innerhalb der Definition werden verschiedene Namensräume verwendet, um die Elemente und Attribute zu kennzeichnen. Die Wahl der Präfixe ist dabei beliebig, nicht normativ und semantisch nicht signifikant (vgl. [OAS07, S.9 f.]). Es werden nur die implementierten Aktivitäten betrachtet. Eventuelle geringfügige Beeinflussungen durch andere Konstrukte sind nicht auszuschließen. Ohne eine offizielle Spezifikation ist es trotz vieler Tests möglich, dass in Spezialsituationen noch zusätzliche Varianten der dargestellten Syntax existieren. Alle Referenzen der Aktivitäten sind unter [Mic12m] und [Mic12n] zu finden.

Im Folgenden werden die verwendeten Namespaces, Basisdatentypen und allgemein referenzierte Definitionen aufgelistet.

```
xmlns = "http://schemas.microsoft.com/netfx/2009/xaml/activities"
xmlns:x = "http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:p = "http://schemas.microsoft.com/netfx/2009/xaml/servicemodel"
```

QNAME = Qualifizierter Name bestehend aus Namespace und lokalem Namen.

NCNAME = Einfaches Literal.

BOOL = Einfacher boolescher Wert.

Eine **activity** kann jede beliebige Aktivität aus den beiden Aktivitätsnamespaces sein (siehe [Mic12m, Mic12n]).

namespace = Eine beliebige Namespace–Definition.

Workflow Deklaration

WF unterstützt zwei verschiedene Möglichkeiten um Xaml-basierte Workflows zu definieren, welche die folgenden Aktivitäten beinhalten. Zum einen normale **Activity**-Workflows und zum anderen servicebasierte **WorkflowServices**.

```
<Activity x:Class="NCNAME" namespace+>
    activity
</Activity>

<p:WorkflowService Name="NCNAME" ConfigurationName="QNAME" namespace+>
    activity
</p:WorkflowService>
```

Die beiden Attribute *x:Name* und *x:Key* können in jeder **activity** und zusätzlich auch in den Kontrollflusskonstrukten des Flowcharts vorkommen. *x:Name* enthält eine eindeutige Bezeichnung in der Form `_ReferenceID1`, über die eine Aktivität referenziert werden kann. *x:Key* beinhaltet den Schlüssel des Falles einer **Switch<T>**- oder **FlowSwitch**-Aktivität, den die Aktivität bedient.

standard-attributes

```
x:Name="NCNAME"?, x:Key="QNAME"?
```

reference = Referenziert eine Aktivität in einem Attribut. Dies wird durch eine Typ-Wert-Kombination erreicht (z.B. `{x:Reference NCNAME}`).

vb-expr = Visual Basic Ausdruck, dessen Ergebnis jeweils vom erwarteten Typ sein muss.

Attribute und Eigenschaftselemente

Alle Aktivitäten besitzen die Möglichkeit die gleiche Konfiguration sowohl durch ein Attribut als auch durch ein sog. *Eigenschaftselement* darzustellen. In jedem Fall kommt nur eine der beiden Varianten zum Einsatz. In Xaml können Objektelemente als Eigenschaftswerte zugewiesen werden. Diese Möglichkeit wird mit der Eigenschaftselementsyntax abgebildet. Die Eigenschaft wird nicht innerhalb des Elementtags als Attribut, sondern mithilfe eines Elements der Form `$ActivityName.PropertyName` angegeben. Für Objekte ist dies zwingend notwendig. Primitive Typen, wie Literale, können sowohl als Attribut als auch als Eigenschaftselement dargestellt werden, auch wenn davon abgeraten wird. Im Folgenden werden nur die in WF tatsächlich aufgetretenen Syntaxausprägungen dargestellt.

```
<$ActivityName>
    <$ActivityName.Trigger>
        XML-Darstellung des Objektelements
    </$ActivityName.Trigger>
</$ActivityName>
```

A XAML(X)-SPEZIFIKATION

Darstellung eines primitiven Typen als Attribut:

```
<$ActivityName Text=" NCNAME " />
```

Äquivalente Darstellung eines primitiven Typen als Eigenschaftselement:

```
<$ActivityName>
  <$ActivityName.Text> NCNAME </$ActivityName.Text>
</ $ActivityName >
```

Variablen

Viele Aktivitäten unterstützen die Deklaration von Variablen in ihrem Gültigkeitsbereich. Der Platzhalter *\$ActivityName* muss hierbei durch den entsprechenden Aktivitätsnamen ersetzt werden.

variables

```
<$ActivityName.Variables>
  <Variable
    x>TypeArguments="QNAME"
    Name=" NCNAME "
    Default="vb-expr"? />+
</ $ActivityName.Variables >
```

Eingabe- und Ausgabeargumente

Datenflüsse in und aus Aktivitäten können mithilfe von Argumenten definiert werden.

inargument

```
<InArgument x>TypeArguments="QNAME"> vb-expr </InArgument>
```

outargument

```
<OutArgument x>TypeArguments="QNAME"> vb-expr </OutArgument>
```

A.1 Primitive Aktivitäten

A.1.1 WriteLine

```
<WriteLine
  Text="vb-expr"?
  TextWriter="vb-expr"?
  DisplayName="NCNAME"?
  standard-attributes />
```

Erwartete Typen:

Text = System.String, **TextWriter** = System.IO.TextWriter

A.1.2 Delay

```
<Delay
  Duration="vb-expr"?
  DisplayName="NCNAME"?
  standard-attributes />
```

Erwartete Typen: **Duration** = System.Timespan

A.1.3 Assign

```
<Assign DisplayName="NCNAME"? standard-attributes>
  <Assign.To> out-argument </Assign.To>
  <Assign.Value> in-argument </Assign.Value>
</Assign>
```

A.1.4 Receive

Die Auslassungspunkte innerhalb der Aktivität signalisieren, dass dort weitere komplexe innere Strukturen, wie z.B. Übergabedaten und Korellationen, enthalten sind. Da diese Aspekte in der Betrachtung jedoch abstrahiert werden, wird auf eine genaue Definition verzichtet.

```
<p:Receive
  OperationName="NCNAME"
  ServiceContractName="QNAME"?
  Action="NCNAME"?
  CanCreateInstance="BOOL"?
  SerializerOption="DataContractSerializer | XmlSerializer"?
  CorrelatesWith="vb-expr"?
  ProtectionLevel="None | Sign | EncryptAndSign"?
  DisplayName="NCNAME"?
  standard-attributes>
  (...)</p:Receive>
```

Erwartete Typen:

CorrelatesWith = System.ServiceModel.Activities.CorrelationHandle

A.1.5 ReceiveReply

Siehe Anhang A.1.4.

```
<p:ReceiveReply  
    Request="reference"  
    Action="NCNAME"?  
    DisplayName="NCNAME"?  
    standard-attributes>  
    (...)  
</p:ReceiveReply>
```

A.1.6 Send

Siehe Anhang A.1.4.

```
<p:Send  
    OperationName="NCNAME"  
    ServiceContractName="QNAME"  
    Action="NCNAME"?  
    CorrelatesWith="vb-expr"?  
    EndpointAddress="vb-expr"?  
    EndpointConfigurationName="NCNAME"?  
    ProtectionLevel="Sign | None | EncryptAndSign"?  
    SerializerOption="DataContractSerializer | XmlSerializer"?  
    TokenImpersonationLevel="None | Anonymous | Identification |  
        Impersonation | Delegation"?  
    DisplayName="NCNAME"?  
    standard-attributes>  
    (...)  
</p:Send>
```

Erwartete Typen:

`CorrelatesWith = System.ServiceModel.Activities.CorrelationHandle,`
`EndpointAddress = System.Uri`

A.1.7 SendReply

Siehe Anhang A.1.4.

```
<p:SendReply  
    Request="reference"  
    Action="NCNAME"?  
    PersistBeforeSend="BOOL"?  
    DisplayName="NCNAME"?  
    standard-attributes>  
    (...)  
</p:SendReply>
```

A.2 Strukturierte Aktivitäten

A.2.1 Sequence

```
<Sequence DisplayName="NCNAME"? standard-attributes>
    variables ?
    activity *
</Sequence>
```

A.2.2 Parallel

Es existiert entweder das Attribut *CompletionCondition* oder das äquivalente Element.

```
<Parallel CompletionCondition="vb-expr"? DisplayName="NCNAME"? standard-attributes>
    variables ?
    <Parallel.CompletionCondition>?
        vb-expr
    </Parallel.CompletionCondition>
    activity *
</Parallel>
```

Erwartete Typen:

CompletionCondition = System.Boolean

A.2.3 Pick

```
<Pick DisplayName="NCNAME"? standard-attributes>
    <PickBranch DisplayName="NCNAME"?*>*
        variables ?
        <PickBranch.Trigger>
            activity
        </PickBranch.Trigger>
        activity ?
    </PickBranch>
</Pick>
```

A.2.4 While

Es existiert entweder das Attribut *Condition* oder das Element <While.Condition>.

```
<While Condition="vb-expr"? DisplayName="NCNAME"? standard-attributes>
    variables ?
    <While.Condition>?
        vb-expr
    </While.Condition>
    activity ?
</While>
```

Erwartete Typen:

Condition = System.Boolean

A.2.5 DoWhile

Es existiert entweder das Attribut *Condition* oder das Element <DoWhile.Condition>.

```
<DoWhile Condition="vb-expr"? DisplayName="NCNAME"? standard-attributes>
    variables ?
    <DoWhile.Condition>?
        vb-expr
    </DoWhile.Condition>
    activity ?
</DoWhile>
```

Erwartete Typen:

Condition = System.Boolean

A.2.6 If

```
<If Condition="vb-expr" DisplayName="NCNAME"? standard-attributes>
    <If.Then>?
        activity
    </If.Then>
    <If.Else>?
        activity
    </If.Else>
</If>
```

Erwartete Typen:

Condition = System.Boolean

A.2.7 Switch<T>

Alle enthaltenen Aktivitäten führen zusätzlich das Attribut *x:Key*, um den bearbeiteten Fallwert zu kennzeichnen. Der *QNAME* des *x:Key*-Attributs kann bei primitiven Datentypen auch ein einfacher Wert sein. Das Element <x:Null> existiert, falls keine Aktivität für den Fallwert angegeben wurde.

```
<Switch
    x>TypeArguments="QNAME"
    Expression="vb-expr"
    DisplayName="NCNAME"? standard-attributes>
    <Switch.Default>?
        activity
    </Switch.Default>
    <x:Null x:Key="QNAME" /*>
        activity *
</Switch>
```

Erwartete Typen:

Expression = T

A.3 Flowchart

Es existiert entweder das Attribut *StartNode* oder das Element <Flowchart.StartNode>. Wenn der Flowchart keine Aktivitäten besitzt, d.h. leer ist, enthält der Knoten <Flowchart.StartNode> das Element <x:Null />.

```
<Flowchart StartNode="reference"? DisplayName="NCNAME"?
    standard-attributes>
    variables ?
    <Flowchart.StartNode>?
        flownode | reference
    </Flowchart.StartNode>
    flownode *
</Flowchart>

flownode
flowstep | flowdecision | flowswitch | ref-element

flowstep
<FlowStep standard-attributes>
    activity
    <FlowStep.Next>?
        flownode
    </FlowStep.Next>
</FlowStep>

ref-elem
<x:Reference>
    NCNAME
    <x:Key>?
        QName
    </x:Key>
</x:Reference>
```

A.3.1 FlowDecision

Es existiert entweder das Attribut *True* (*False*) oder das Element <FlowDecision.True> (<FlowDecision.False>).

```
<FlowDecision Condition="vb-expr" True="reference"? False="reference"?
    standard-attributes>
    <FlowDecision.True>?
        flownode
    </FlowDecision.True>
    <FlowDecision.False>?
        flownode
    </FlowDecision.False>
</FlowDecision>
```

Erwartete Typen:

Condition = System.Boolean

A.3.2 FlowSwitch<T>

Alle von `FlowSwitch<T>` referenzierten Aktivitäten oder das Element `<x:Reference>` führen zusätzlich das Attribut *x:Key*, um den Fallwert zu kennzeichnen. Es existiert entweder das Attribut *Default* oder das Element `<FlowSwitch.Default>`.

```
<FlowSwitch Default="reference"?
    standard-attributes>
    <FlowSwitch.Default>?
        flownode
    </FlowSwitch.Default>
    flownode *
</FlowSwitch>
```

Erwartete Typen:

`Expression` = T

A.4 StateMachine

Es existiert entweder das Attribut *InitialState* oder das äquivalente Element.

```
<StateMachine
    InitialState="reference"?
    DisplayName="NCNAME"?>
    variables ?
    (
    state
    <x:Reference*>
        NCNAME
    </x:Reference>
    |
    <StateMachine.InitialState>
        state
    </StateMachine.InitialState>
    <x:Reference*>
        NCNAME
    </x:Reference>
    )
</StateMachine>
```

Wenn *isFinal* wahr ist, dann darf `<State>` nur das Element `<State.Entry>` enthalten.

```
state

<State x:Name="NCNAME" isFinal="BOOL"? DisplayName="NCNAME"?>
    variables ?
    <State.Entry>?
        activity
    </State.Entry>
    <State.Exit>?
        activity
    </State.Exit>
    <State.Transitions>?
        transition +
    </State.Transitions>
</State>
```

A XAML(X)-SPEZIFIKATION

Je State darf nur eine Transition ohne *Condition* und *Trigger* existieren. Im Falle eines Shared Triggers, dessen Aktivität mehrere Transitions aktiviert, wird die entsprechende Trigger-Aktivität über das Attribut *x:Name* referenziert. Es existieren wiederum nur die Attribute *Condition*, *To*, *Trigger* oder deren äquivalente Elemente.

transition

```
<Transition To="reference" Trigger="reference"? Condition="vb-expr"?>
  <Transition.Trigger>?
    activity
  </Transition.Trigger>
  <Transition.To>?
    state | <x:Reference>NCNAME</x:Reference>
  </Transition.To>
  <Transition.Action>?
    activity
  </Transition.Action>
  <Transition.Condition>?
    vb-expr
  </Transition.Condition>
</Transition>
```

Erwartete Typen:

Condition = System.Boolean

Literatur

- [ACKM04] ALONSO, G., F. CASATI, H. KUNO und V. MACHIRAJU: *Web services: concepts, architectures and applications*. Springer Verlag, 2004. ISBN: 978-3540440086.
- [ALSU07] AHO, A.V., M.S. LAM, R. SETHI und J.D. ULLMAN: *Compilers: principles, techniques, & tools*. Pearson Addison-Wesley, 2007. ISBN: 978-0321491695.
- [Apa12a] APACHE ODE: *Architectural Overview*, 2012. Online verfügbar unter <http://ode.apache.org/architectural-overview.html>; letzter Abruf am 29. Juni 2012.
- [Apa12b] APACHE ODE: *BPEL 2.0 Compliance*, 2012. Online verfügbar unter <http://ode.apache.org/ws-bpel-20-specification-compliance.html>; letzter Abruf am 29. Juni 2012.
- [Apa12c] APACHE ODE: *Creating a Process*, 2012. Online verfügbar unter <http://ode.apache.org/creating-a-process.html>; letzter Abruf am 29. Juni 2012.
- [Buk08] BUKOVICS, B.: *Pro WF: Windows Workflow in .NET 3.5*. Apress, Juni 2008. ISBN: 978-1430209751.
- [Buk10] BUKOVICS, B.: *Pro WF: Windows Workflow in .NET 4*. Apress, Juni 2010. ISBN: 978-1430227212.
- [DvT05] DUMAS, M., W. M. P. VAN DER AALST und A. H. TER HOFSTEDE: *Process-aware information systems: bridging people and software through process technology*. Wiley-Blackwell, 2005. ISBN: 978-0-471-66306-5.
- [GMW08] GIERDS, C., A.J. MOOIJ und K. WOLF: *Specifying and generating behavioral service adapters based on transformation rules*. Technischer Bericht, Humboldt-Universität zu Berlin, Institut für Informatik, 2008.
- [GMW10] GIERDS, C., A. J. MOOIJ und K. WOLF: *Reducing Adapter Synthesis to Controller Synthesis*. IEEE Transactions on Services Computing, 99:72–85, 2010.
- [HR04] HARDY, S. und P.N. ROBILLARD: *Modeling and simulation of molecular biology systems using petri nets: modeling goals of various approaches*. J Bioinform Comput Biol, 2(4):595–613, 2004.
- [Jac12] JACOBS, R.: *Windows Workflow Foundation (WF4) – Introduction to State Machine Hands On Lab*, April 2012. Online verfügbar unter <http://code.msdn.microsoft.com/windowsdesktop/Windows-Workflow-b4b808a8>; letzter Abruf am 29. Juni 2012.
- [KMWL09] KOPP, O., D. MARTIN, D. WUTKE und F. LEYMANN: *The Difference Between Graph-Based and Block-Structured Business Process Modelling Languages*. Enterprise Modelling and Information Systems, 4(1):3–13, 2009.

- [Len11] LENHARD, J.: *A Pattern-based Analysis of WS-BPEL and Windows Workflow*. Technischer Bericht 88, Fakultät Wirtschaftsinformatik und Angewandte Informatik, 2011.
- [LK08] LOHMANN, N. und J. KLEINE: *Fully-automatic Translation of Open Workflow Net Models into Simple Abstract BPEL Processes*. In: *Modellierung 2008, 12.-14. März 2008, Berlin, Proceedings*, Band P-127 der Reihe *Lecture Notes in Informatics (LNI)*, Seiten 57–72. GI, 2008.
- [Loh07] LOHMANN, N.: *A Feature-Complete Petri Net Semantics for WS-BPEL 2.0 and its Compiler BPEL2oWFN*. Informatik-Berichte 212, Humboldt-Universität zu Berlin, August 2007.
- [Loh10] LOHMANN, N.: *Petri Net API*, Juli 2010. Online verfügbar unter <http://download.gna.org/service-tech/pnapi/pnapi.pdf>; letzter Abruf am 29. Juni 2012.
- [Mic06] MICROSOFT: *Microsoft Open Specification Promise*, November 2006. Online verfügbar unter <http://msdn.microsoft.com/de-de/library/bb979152.aspx>; letzter Abruf am 29. Juni 2012.
- [Mic07] MICROSOFT: *Übersicht über Workflows*, 2007. Online verfügbar unter <http://msdn.microsoft.com/de-de/library/aa349006.aspx>; letzter Abruf am 29. Juni 2012.
- [Mic12a] MICROSOFT: *Adding a Service Reference in a Workflow Solution*, 2012. Online verfügbar unter <http://msdn.microsoft.com/en-us/library/gg281653.aspx>; letzter Abruf am 29. Juni 2012.
- [Mic12b] MICROSOFT: *Architektur von Windows-Workflows*, 2012. Online verfügbar unter <http://msdn.microsoft.com/de-de/library/dd489413.aspx>; letzter Abruf am 29. Juni 2012.
- [Mic12c] MICROSOFT: *Configuring System-Provided Bindings*, 2012. Online verfügbar unter <http://msdn.microsoft.com/en-us/library/ms731092.aspx>; letzter Abruf am 29. Juni 2012.
- [Mic12d] MICROSOFT: *Data Member Order*, 2012. Online verfügbar unter <http://msdn.microsoft.com/en-us/library/ms729813.aspx>; letzter Abruf am 29. Juni 2012.
- [Mic12e] MICROSOFT: *Extensible Application Markup Language (XAML)*, März 2012. Online verfügbar unter <http://www.microsoft.com/en-us/download/details.aspx?id=19600>; letzter Abruf am 29. Juni 2012.
- [Mic12f] MICROSOFT: *Introducing XML Serialization*, 2012. Online verfügbar unter <http://msdn.microsoft.com/en-us/library/182eeyhh.aspx>; letzter Abruf am 29. Juni 2012.
- [Mic12g] MICROSOFT: *Konfigurieren von Diensten mit Konfigurationsdateien*, 2012. Online verfügbar unter <http://msdn.microsoft.com/de-de/library/ms733932.aspx>; letzter Abruf am 29. Juni 2012.

- [Mic12h] MICROSOFT: *.NET Context Exchange Protocol Specification*, März 2012. Online verfügbar unter [http://msdn.microsoft.com/en-us/library/cc441982\(PROT.10\).aspx](http://msdn.microsoft.com/en-us/library/cc441982(PROT.10).aspx); letzter Abruf am 29. Juni 2012.
- [Mic12i] MICROSOFT: *.NET Framework 4 Platform Update 1 State Machine Workflows*, 2012. Online verfügbar unter <http://msdn.microsoft.com/en-us/library/ee264171.aspx>; letzter Abruf am 29. Juni 2012.
- [Mic12j] MICROSOFT: *.NET Framework Conceptual Overview*, 2012. Online verfügbar unter <http://msdn.microsoft.com/en-us/library/zw4w595w.aspx>; letzter Abruf am 29. Juni 2012.
- [Mic12k] MICROSOFT: *Pick Activity*, 2012. Online verfügbar unter <http://msdn.microsoft.com/en-us/library/ee358746.aspx>; letzter Abruf am 29. Juni 2012.
- [Mic12l] MICROSOFT: *Properties (C# Programming Guide)*, 2012. Online verfügbar unter [http://msdn.microsoft.com/en-us/library/x9fsa0sw\(v=vs.100\).aspx](http://msdn.microsoft.com/en-us/library/x9fsa0sw(v=vs.100).aspx); letzter Abruf am 29. Juni 2012.
- [Mic12m] MICROSOFT: *System.Activities.Statements Namespace*, 2012. Online verfügbar unter <http://msdn.microsoft.com/en-us/library/System.Activities.Statements>; letzter Abruf am 29. Juni 2012.
- [Mic12n] MICROSOFT: *System.ServiceModel.Activities Namespace*, 2012. Online verfügbar unter <http://msdn.microsoft.com/en-us/library/system.servicemodel.activities>; letzter Abruf am 29. Juni 2012.
- [Mic12o] MICROSOFT: *Understanding the Activity State Model*, 2012. Online verfügbar unter [http://msdn.microsoft.com/en-us/library/aa718190\(v=vs.90\)](http://msdn.microsoft.com/en-us/library/aa718190(v=vs.90)); letzter Abruf am 29. Juni 2012.
- [Mic12p] MICROSOFT: *Variables and Arguments*, 2012. Online verfügbar unter <http://msdn.microsoft.com/en-us/library/dd489456.aspx>; letzter Abruf am 29. Juni 2012.
- [Mic12q] MICROSOFT: *Vom Datenvertragsserialisierer unterstützte Typen*, 2012. Online verfügbar unter <http://msdn.microsoft.com/de-de/library/ms731923>; letzter Abruf am 29. Juni 2012.
- [Mic12r] MICROSOFT: *Xaml Object Mapping Specification 2009*, April 2012. Online verfügbar unter <http://www.microsoft.com/en-us/download/details.aspx?id=19600>; letzter Abruf am 29. Juni 2012.
- [MRS05] MASSUTHE, P., W. REISIG und K. SCHMIDT: *An Operating Guideline Approach to the SOA*. Annals of Mathematics, Computing & Teleinformatics, 1(3):35–43, 2005.
- [Mur89] MURATA, T.: *Petri nets: Properties, analysis and applications*. Proceedings of the IEEE, 77(4):541–580, 1989.
- [OAS07] OASIS: *Web Services Business Process Execution Language Version 2.0*, April 2007. Online verfügbar unter <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf>; letzter Abruf am 29. Juni 2012.

- [OAS11] OASIS: *Universal Business Language Version 2.1 - Committee Specification Draft 02 / Public Review Draft 02*, Mai 2011. Online verfügbar unter <http://docs.oasis-open.org/ubl/prd2-UBL-2.1/UBL-2.1.pdf>; letzter Abruf am 29. Juni 2012.
- [OVVDA⁺07] OUYANG, C., E. VERBEEK, W.M.P. VAN DER AALST, S. BREUTEL, M. DUMAS und A.H. TER HOFSTEDE: *Formal semantics and analysis of control flow in WS-BPEL*. Science of Computer Programming, 67(2):162–198, 2007.
- [Pel03] PELTZ, C.: *Web services orchestration and choreography*. Computer, 36(10):46–52, 2003.
- [RR98] REISIG, W. und G. ROZENBERG: *Lectures on Petri nets II: applications*. Springer, 1998. ISBN: 978-3-540-65307-3.
- [RtEv05] RUSSELL, N., A. H. TER HOFSTEDE, D. EDMOND und W. M. P. VAN DER AALST: *Workflow data patterns: identification, representation and tool support*. In: *Proceedings of the 24th international conference on Conceptual Modeling*, ER'05, Seiten 353–368, Berlin, Heidelberg, 2005. Springer-Verlag.
- [SV09] SHETTY, S. und S. VADIVEL: *Interoperability issues seen in Web Services*. International Journal of Computer Science and Network Security (IJCSNS), 9(8):160–168, 2009.
- [Van98] VAN DER AALST, W.M.P.: *The application of Petri nets to workflow management*. Journal of Circuits Systems and Computers, 8:21–66, 1998.
- [VMSW09] VAN DER AALST, W. M. P., A. MOOIJ, C. STAHL und K. WOLF: *Service Interaction: Patterns, Formalization, and Analysis*. In: *Formal Methods for Web Services*, Band 5569 der Reihe *Lecture Notes in Computer Science*, Seiten 42–88. Springer Berlin/ Heidelberg, 2009.
- [Vv04] VAN DER AALST, W. M. P. und K.M. VAN HEE: *Workflow management: models, methods, and systems*. The MIT press, 2004. ISBN: 0-262-72046-9.
- [W3C99] W3C: *Hypertext Transfer Protocol – HTTP/1.1*, Juni 1999. Online verfügbar unter <http://tools.ietf.org/html/rfc2616>; letzter Abruf am 29. Juni 2012.
- [W3C01] W3C: *Web Services Description Language (WSDL) 1.1*, März 2001. Online verfügbar unter <http://www.w3.org/TR/wsdl>; letzter Abruf am 29. Juni 2012.
- [W3C04] W3C: *Web Services Architecture*, Februar 2004. Online verfügbar unter <http://www.w3.org/TR/ws-arch/>; letzter Abruf am 29. Juni 2012.
- [W3C07a] W3C: *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*, April 2007. Online verfügbar unter <http://www.w3.org/TR/soap12-part1/>; letzter Abruf am 29. Juni 2012.
- [W3C07b] W3C: *Web Services Addressing*, September 2007. Online verfügbar unter <http://www.w3.org/standards/techs/wsaddr>; letzter Abruf am 29. Juni 2012.

LITERATUR

- [W3C08] W3C: *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, November 2008. Online verfügbar unter <http://www.w3.org/TR/REC-xml/>; letzter Abruf am 29. Juni 2012.
- [WFM99] WFMC: *Workflow Management Coalition Terminology & Glossary (WFMC-TC-1011)*. Technischer Bericht, Workflow Management Coalition, Februar 1999.
- [Wol09] WOLF, K.: *Does My Service Have Partners?* In: *Transactions on Petri Nets and Other Models of Concurrency II*, Band 5460 der Reihe *Lecture Notes in Computer Science*, Seiten 152–171. Springer Berlin/ Heidelberg, 2009.